





2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Módulo 3 – Desarrollo Front End dinámico

Índice

Introducción

Framework vs Librería 3

¿Qué es Back-end? 5

¿Qué es Front-end? 6

Single Web Applications

Single Web Applications 7

Modelo Vista Controlador (MVC) 14

Presentación de Angular

¿Por qué Angular? 18

Instalación de Angular 19

Estructura de un proyecto Angular 20

Explorando las piezas de Angular

Arquitectura Angular 23

Módulos en Angular 24

Componentes en Angular 26

Templates, Plantillas o Vistas en Angular 31

Expresiones y Pipes en Angular 33

Directivas en Angular 34

Databinding en Angular 38

Servicios en Angular 41

Sistema de Routing en Angular 44

Trabajando con Angular

Sesión de Usuario con Angular 52

Formularios en Angular 59

Testing en Angular 66

Primera app con Angular

Crear Aplicación con Angular CLI 75







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Framework vs Libreria

En este módulo veremos las diferencias entre una librería y un framework ya que es importante entender porque a veces se habla de una cosa o la otra, y qué alcances tienen en cada caso para nuestro trabajo como desarrolladores. Es importante de aclarar que, si googleas, es común encontrar en este contexto, la Biblioteca como sinónimo de Librería

¿Qué es una librería?

Una librería normalmente proporciona una serie de funciones y métodos muy concretos para simplificar tareas complejas. Estas funciones y métodos se pueden utilizar sin necesidad de adaptar/modificar nuestra estructura de aplicación.

Un ejemplo podría ser una librería matemática que ofrece funciones avanzadas para cálculos estadísticos. JQuery es otro ejemplo, es una librería que proporciona funciones sencillas para manejo del DOM, para comunicación AJAX y algunas otras utilidades.

Otro ejemplo con jQuery sería si se quisiera realizar un efecto de fade (desvanecer) sobre un elemento del DOM en vez de desarrollar un código en CSS y/o javascript para lograr el objetivo, simplemente podemos usar el método ofrecido por la librería

$(*selector*).fadeIn(*velocidad, función de callback*);

¿Qué es un Framework?

Los frameworks son la base sobre la cual los desarrolladores crean programas para plataformas específicas. Estos están diseñados para disminuir la cantidad de problemas que enfrenta un programador durante el desarrollo. Los framework pueden tener funciones y objetos definidos o no definidos que el programador puede usar o sobrescribir para crear una aplicación. **La tarea principal de un framework es proporcionar un código y metodología de trabajo estandarizado que se pueda aplicar a una variedad de proyectos de aplicaciones.** Los Framework tienen un alcance más amplio e incluyen casi todo lo necesario para hacer una aplicación. Algunos de los framework populares en Javascript son Node.js y AngularJS.

Libreria vs Framework: la diferencia clave

Tanto la librería como el framework desempeñan un papel vital en el desarrollo de software. Una librería realiza una operación específica o bien definida, mientras

que un framework proporciona un *esqueleto* donde los desarrolladores definen el contenido de la aplicación de la operación.

Una de las diferencias clave es la **Inversión de control**. Cuando llamamos a una función o un método desde una librería, se tiene el control. Con un framework se invierte la situación. En la mayoría de los casos, el framework solo proporciona el concepto. El trabajo de la aplicación es definir mejor la funcionalidad para los usuarios finales.

En una librería el código del desarrollador es el que está al mando y utiliza las funciones de la librería cuándo y cómo quiere. En un *framework*, éste es el que está al mando y el código desarrollado debe encajarse en su estructura y normas.

Otra gran diferencia es que un framework utiliza librerías para facilitar las tareas de desarrollo y las propias funcionalidades que nos ofrece el framework.

En resumen

FRAMEWORK VS LIBRERÍA

Conjunto de herramientas que trabajan en un proyecto completo bajo ciertas reglas.

Herramienta con una sola utilidad específica.

Tiene funcionalidades integradas para que no necesites librerías externas.

La compatibilidad de sus funcionalidades está asegurada.

El framework define la forma en que debes desarrollar el proyecto.

Eres libre de usar las librerías que desees en la estructura que quieras.

Debes controlar la compatibilidad de cada libreria con las demás.

Puedes usar varias librerías según tus necesidades.

Laravel

EDgrid

Aprende a programar con las librerías y frameworks más populares en:
ed.team/cursos

EDteam

[Libreria vs framework. (2021)].

¿Que es Back-end?

Backend

El **Backend** es aquello que ***se encuentra del lado del servidor*** y se encarga de *interactuar con bases de datos, verificar maniobras de sesiones de usuarios, montar la página en un servidor y servir todas las vistas creadas por el desarrollador frontend*.

En este caso el número de tecnologías es mucho menos limitado, puesto que la programación backend puede alcanzar *lenguajes como PHP, Python, .NET, Java, etc*., y las bases de datos sobre las que se trabaja pueden ser SQL, MongoDB, MySQL, entre otras.

La idea de esta abstracción es mantener separadas las diferentes partes de un sistema web o software para tener un mejor control. En pocas palabras, el objetivo es que el frontend recoja los datos y el backend los procese.

Estas dos capas que forman una aplicación web son independientes entre sí (no comparten código), pero intercambian información. Esta división permite que el acceso a las bases de datos solo se haga desde el backend y el usuario no tenga acceso al código de la aplicación, mientras que la programación del lado del cliente permite que el navegador pueda, por ejemplo, controlar dónde el usuario hace clic o acceder a sus ficheros.

Con esta separación de entornos el usuario de una aplicación web lo que hace es, por ejemplo, iniciar sesión escribiendo su usuario y contraseña en un formulario; a continuación, los datos se envían y el backend toma esta información que viene desde el HTML y busca las coincidencias de usuario en la base de datos con una serie de procesos invisibles para el usuario. En este punto, el servidor mandaría un mensaje al frontend dándole acceso (o no) a la aplicación.

Lenguajes Web Backend

Encontramos varios, por ejemplo, PHP, Python, Rails, Go, C#, Java, Node JS (JavaScript) entre otros. Como vemos, mientras que para el frontend se acostumbra a trabajar solo con tres lenguajes, en el backend hay unos cuantos más. Por suerte, un desarrollador backend no necesita saberlos todos.

Quizás notaste que tenemos JavaScript tanto por el lado del cliente como por el lado del servidor. JavaScript se creó originalmente como lenguaje para el frontend, pero en los últimos años se ha creado su lugar dentro del backend con NodeJS, un motor que interpreta JavaScript en el servidor sin necesidad de un navegador. Esto no quiere decir que el JavaScript que tenemos en el cliente tenga alguna conexión con el que se encuentra en el servidor: cada uno corre por su parte de manera independiente. El JavaScript del cliente corre en el navegador y no tiene ningún enlace ni ninguna conexión con el que hay en el servidor y no le interesa saber cómo está montada la arquitectura del servidor ni cómo se conecta a la base de datos.

Ahora se puede utilizar el mismo lenguaje en todos los contextos del desarrollo: JavaScript en el cliente de escritorio (DOM), en el cliente móvil (Cordova, React Native), en el servidor (Node.js) o en la BBDD (MongoDB). La posibilidad de trabajar frontend y backend con un mismo lenguaje desde el punto de vista del desarrollador es muy cómoda, especialmente para aquellos que trabajan ambos mundos.

En cuanto a la tecnología, las herramientas que se utilizan en el backend son: editores de código, compiladores, algunos depuradores para revisar errores y seguridad, gestores de bases de datos entre otras cosas.







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

¿Qué es Front-end?

Introducción

Ya se ha mencionado que un desarrollador Full Stack es un perfil híbrido que puede desenvolverse tanto en el front-end (parte visual) como en el back-end (parte lógica) de un desarrollo web. Un Full Stack Developer cuenta con una faceta integral y posee conocimientos y herramientas que le permiten afrontar cualquiera de las etapas de un proyecto web. Los desarrolladores web lo hacen a través de diversos lenguajes de programación. El lenguaje que usan en cada momento depende del tipo de tarea que están haciendo. Pero ¿qué es exactamente Frontend?

Frontend

El **Frontend** son aquellas tecnologías de desarrollo ***web del lado del cliente,*** es decir, las que corren en el navegador del usuario y que son básicamente tres las que veremos: *HTML, CSS y JavaScript*. El Desarrollador Frontend es responsable de la interacción directa del usuario, por lo que se desarrolla cuidando el lado más visual de las aplicaciones, como el cuidado de colores, botones, enlaces, menús y todo lo que vemos en una página cuando estamos accediendo. Precisamente, un profesional frontend necesita tener un ojo constante para la mejor experiencia de usuario

El frontend *se enfoca en el usuario*, en todo con lo que puede interactuar y lo que ve mientras navega. Una buena experiencia de usuario, inmersión y usabilidad son algunos de los objetivos que busca un buen desarrollador frontend, y hoy en día hay una gran variedad de **Frameworks**, preprocesadores y librerías que ayudan en esta tarea.

Lenguajes Web Frontend

A pesar de que hay varios lenguajes que se usan en el frontend, nosotros nos basaremos en tres, HTML, CSS y JavaScript, aunque HTML y CSS no son lenguajes de programación, no se debe confundir lenguajes de programación como ejemplo JavaScript, ActionScript o Java, con lenguajes de marcado como HTML o lenguaje de hojas de estilo cómo CSS. También existen otros lenguajes frontend, como por ejemplo ActionScript, Java, Silverlight, VBScript u otros lenguajes XML, pero se usan poco en comparación con HTML, CSS y JavaScript.

HTML es un lenguaje de marcado de los contenidos de un sitio web, es decir, para designar la función de cada elemento dentro de la página: titulares, párrafos, listas, tablas, etc. Es el esqueleto de la web y la base de todo el frontend.

CSS es un lenguaje de hojas de estilo creado para controlar la presentación de la página definiendo colores, tamaños, tipos de letras, posiciones, espaciados, etc.

También existen frameworks para CSS muy famosos como Bootstrap y foundation, los cuales aportan mucho para la creación de UI (interfaz de usuario).

JavaScript es un lenguaje de programación interpretado que se encarga del comportamiento de una página web y de la interactividad con el usuario.

Aparte, junto al cliente también tenemos los frameworks (Angular o React), las librerías, los preprocesadores, los plugins... pero todo gira alrededor de HTML, CSS y JavaScript.

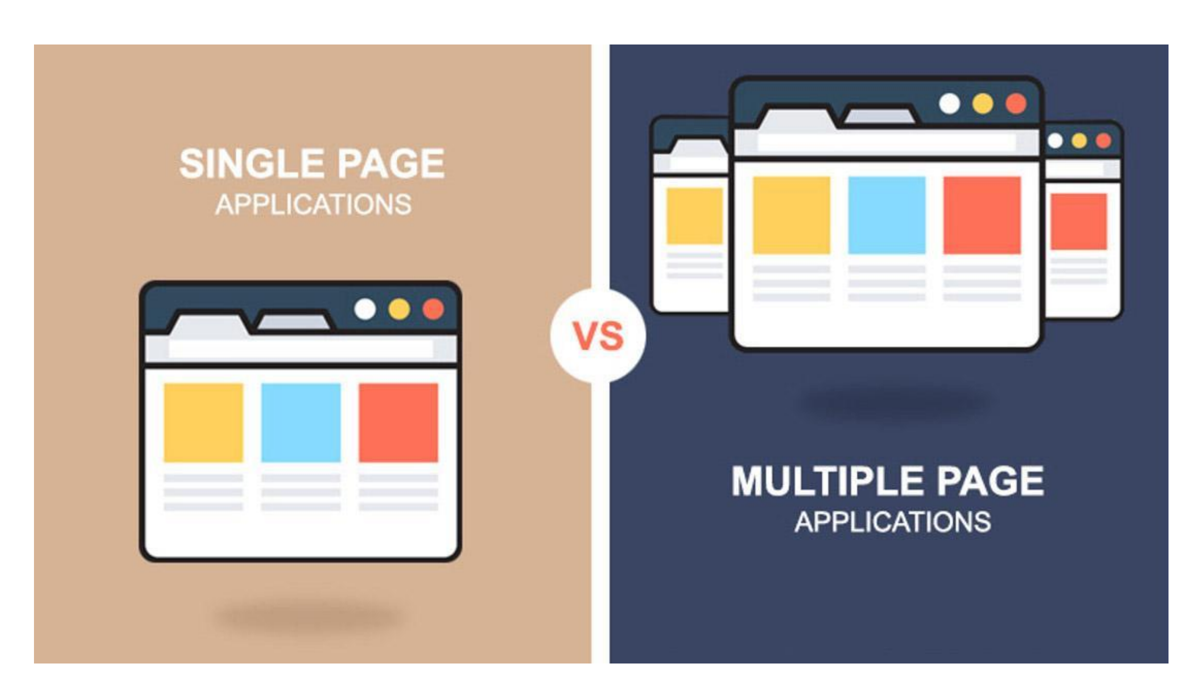
Single Web Applications

Conoceremos las dos arquitecturas más usadas para el desarrollo de páginas webs a fin de entender el nuevo paradigma de las aplicaciones webs. La evolución de la tecnología y los dispositivos desde los cuales nos conectamos a internet han marcado un punto de inflexión, abriendo paso a una nueva forma de construir e interactuar con las páginas webs.

¿Qué es una aplicación SPA?

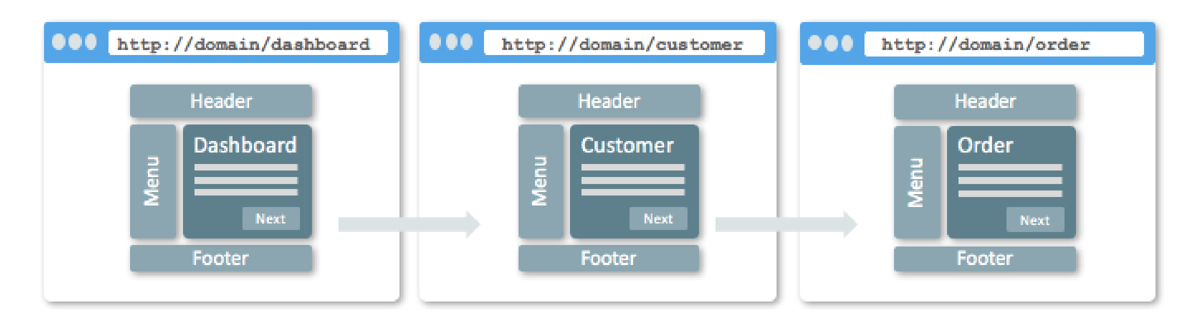
Las Multi Page Application o MPA hacen referencia a Arquitecturas Web Clásicas en donde uno dispone de múltiples páginas HTML y cada una carga diferentes contenidos apoyándose en la navegación contra el servidor. Es decir, cada página muestra su contenido y se conecta mediante links con las demás y todas son generadas desde el servidor.

A diferencia de las tradicionales MPA (múltiple page application), las aplicaciones SPA (single page application) consisten en aplicaciones de una sola página por lo que todas vistas de la misma se generan dinámicamente gracias a la capacidad de javascript para manipular el DOM. De esta manera, no recarga el navegador cada vez que el usuario hace una petición lo que permite que ésta sea óptima en rendimiento, mantenimiento y escalabilidad.



SPA vs MPA. (2021).

Es decir, que en las aplicaciones SPA, encuentras toda la funcionalidad que hace a una aplicación completa y en una única página web (index.html). Esto es posible, gracias al sistema de ruteo que propone tanto Angular como otros frameworks de desarrollo Frontend, el cual permite la generación dinámica de la página web.

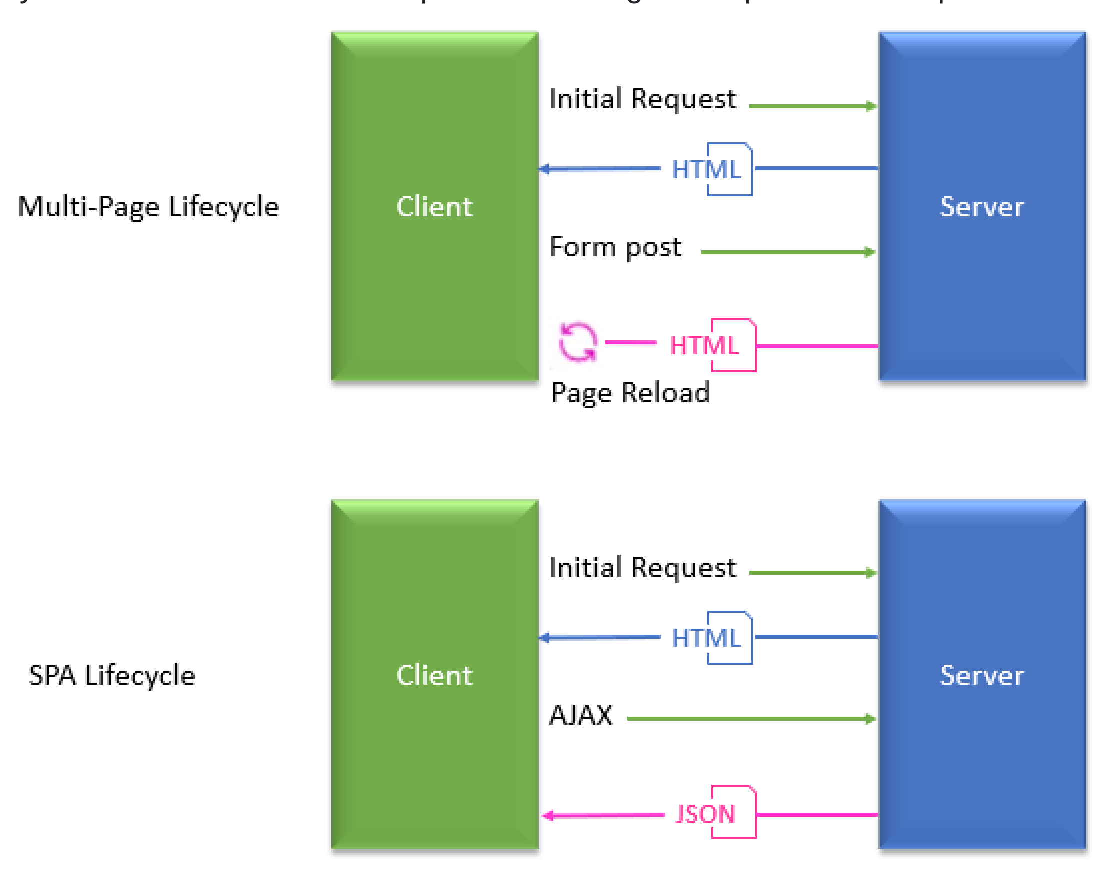


Spa content update. (n.d.).

En dicha imagen, podemos observar la generación dinámica de la página index.html dónde dashboard, customer y orders, que son secciones dentro del DOM, son generados dinámicamente según la petición del usuario.

En la siguiente imagen podemos clarificar un poco más cómo funciona la lógica de consultas en cada caso. Observemos que en el caso de la MPA luego de recibir el HTML se generan consultas POST que también podrían ser GET, y el servidor retorna un nuevo HTML. Mientras que, en la SPA al llegar el primer HTML desde el

servidor, las siguientes llamadas se realizan mediante AJAX y que el servidor retorna un JSON. Eso se refleja del lado del cliente, como se dijo antes, en que no haya un refresco de la web completa sino de algún componente en específico.



Spa content update. (n.d.)

En la imagen, podemos observar la generación dinámica de la página index.html dónde dashboard, customer y orders, que son secciones dentro del DOM, generados dinámicamente según la petición del usuario.

Características funcionales de una SPA:

Las SPA tiene 3 detalles funcionales que las caracterizan y son:

1. **Punto de entrada central**: Un punto de entrada único que se genera dinámicamente según la petición del usuario.
2. **Página fija, Vistas cambiantes**: Como en el caso de una aplicación de escritorio, nos mantenemos en un “marco único” y fijo, mientras que “vistas dinámicas” van ofreciéndonos las distintas posibilidades del uso y navegación.
3. **Página fija, no URL fija**: Es posible que la dirección URL sufra cambios en base a las actividades de uso de la plataforma y vaya modificándose, aunque ese “marco único” se mantenga fijado. Esto es un tanto

reduccionista (existen SPA que no transforman sus direcciones), pero es útil para comprender su mecánica.

Viajar ligera de equipaje

En las SPA, las peticiones cliente-servidor tienden a ser más laxas y livianas que en las aplicaciones tradicionales MPA dado que sólo consisten en la transmisión de datos, sólo datos. Es importante agregar en este punto, que muchos procesos quedan del lado del cliente (navegador web) gracias a herramientas que provee Angular tales como el LocalStorage.

No todos es color de rosas, algunas desventajas del enfoque SPA son:

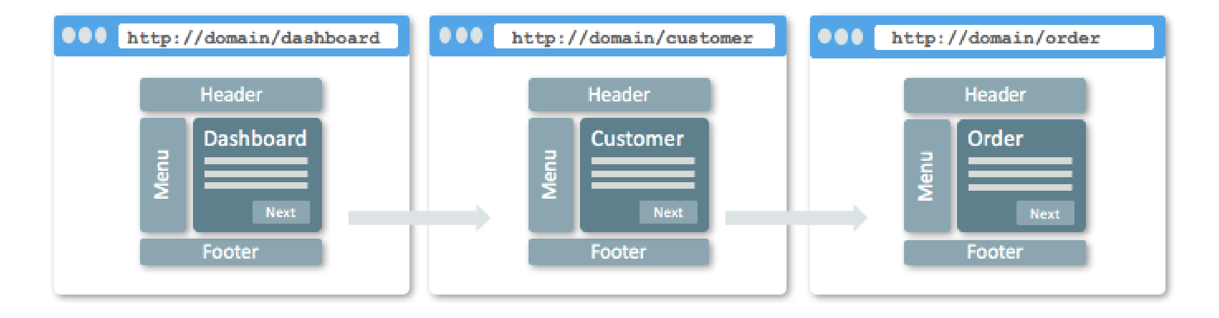
- Se pierde el SEO, aunque hay formas de evitarlo pero no es algo natural en un SPA ya que los robots no leen o interpretan contenido cargado en segundo plano.
- La primera carga puede tardar un poco ya que el peso del proyecto estará en función al tamaño de este y también existen estrategias para resolver este problema.
- Curva de aprendizaje, nos obliga aprender nuevos frameworks o librerías para evaluar la más adecuada para nuestros proyectos.

MPA Híbridas

En muchas ocasiones también nos podemos encontrar con situaciones de hibridación en la cual disponemos de una aplicación MPA. Pero que contiene características de SPA cómo puede ser la construcción de componentes con React.js aunque mantiene el enfoque multipágina.

Varias vistas, no varias páginas

Las aplicaciones SPA consisten de un único documento HTML y muchas vistas ya que existen distintas partes o componentes, que controlan y definen una parte en pantalla o vista. Por ejemplo, en la siguiente imagen, los componentes individuales definen y controlan cada una de las vistas como sigue:



Spa content update. (n.d.).

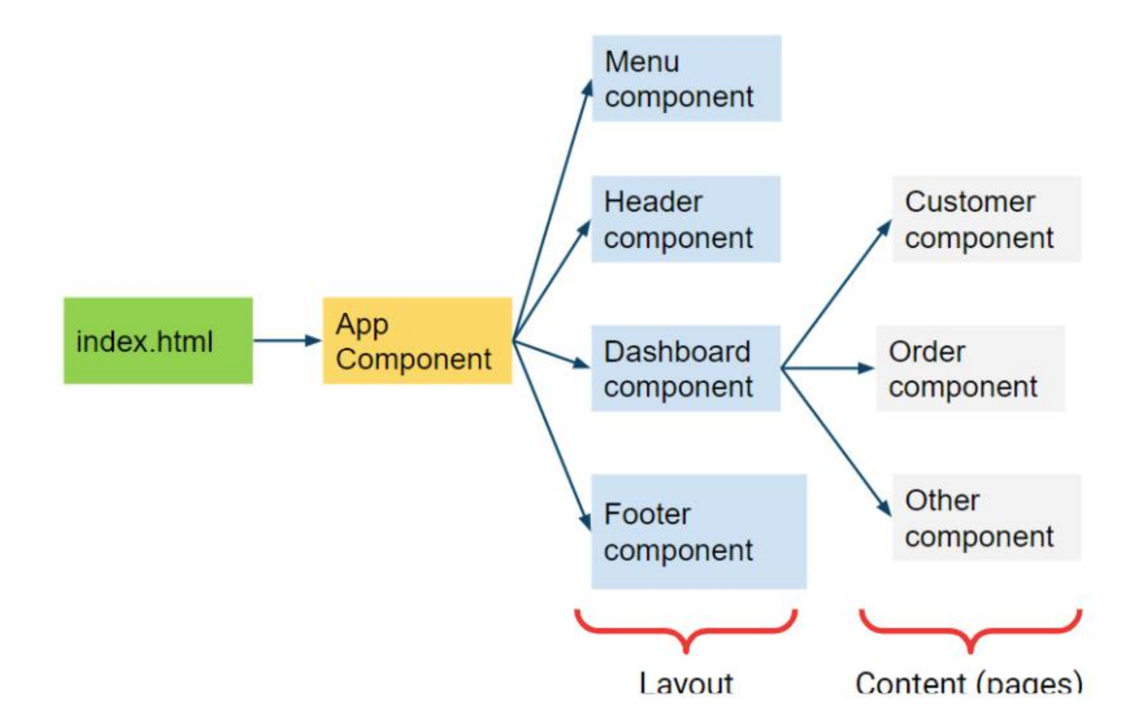






2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Entonces del ejemplo anterior podemos inferir que existe una jerarquía de componentes tal y como se muestra en la siguiente imagen:



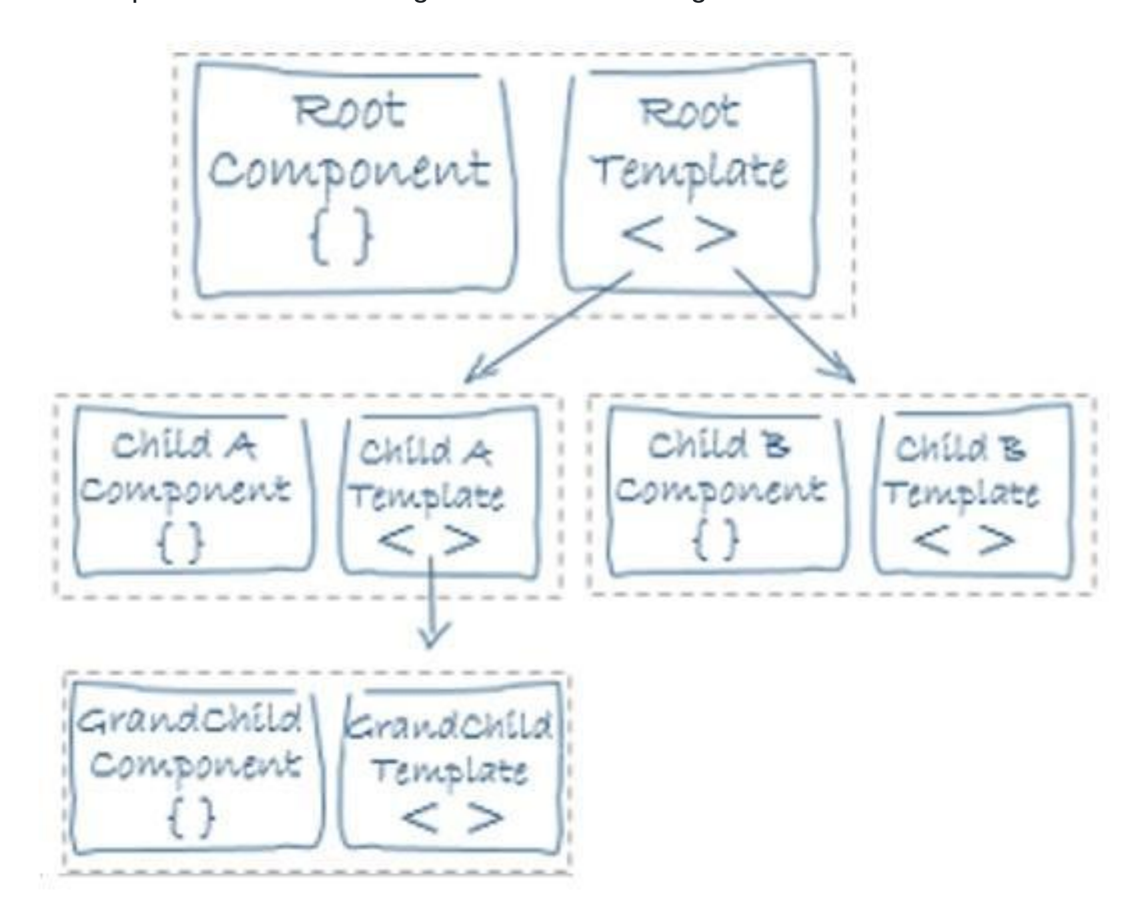
Plantillas y vistas

Ahora los componentes requieren de un template o plantilla html para que las vistas puedan ser interpretadas por el navegador web. Un template es un bloque HTML que le dice al framework, en este caso Angular, cómo renderizar o dibujar el componente definiendo así la vista. El template es código HTML dotado de la capacidad de interpretar o agregar sintaxis propia de cada framework, en Angular, por ejemplo la inclusión de lógica y bucles (ifs y fors), el sistema de binding que veremos más adelante y que permite enlazar las variables al código HTML.



Angular Template. (n.d.).

Como se dijo antes, si tenemos una jerarquía de componentes, también podemos pensar que existe una jerarquía de vistas, las cuales contienen vistas embebidas o anidadas, o incluso dentro de otros componentes. Es importante empezar a pensar cómo un proyecto es un árbol de componentes y vistas que se relacionan entre sí. Luego veremos que, a su vez, podemos agrupar estos componentes y vistas en módulos para su reutilización gracias al uso de Angular.



Árbol de componentes. (n.d.).

¿En qué lenguaje de programación se hacen las SPA?

Una SPA se creará siempre en Javascript. Ya que, al ser una aplicación web ejecutada del lado del cliente, no hay otro lenguaje que pueda hacer eso. A esto habrá que sumarle, lógicamente, el HTML y CSS.

Dentro de Javascript, contaremos con multitud de librerías y frameworks que nos facilitarán el desarrollo de las SPA. Algunas de las más usadas son:

- AngularJS
- Angular
- VUE
- React
- EmberJS
- Polymer

Este listado son las librerías y frameworks más comunes, aunque hay muchos otros. Escoger una librería y un framework dependerá, básicamente, de tus gustos, la experiencia que tengas con los mismos, el tamaño de la aplicación a desarrollar…

De todos modos, esto sólo aplica al front-end. Si queremos ampliar la pregunta al lado del back-end, es decir, del servidor, nos serviría cualquier lenguaje de back-end para producir la parte del servidor. Lo que tendríamos que hacer es crear un API REST que devuelve el JSON necesario para alimentar de datos a la SPA. Es decir, la SPA nos dará igual cómo esté desarrollada del lado del back-end, es totalmente independiente a este.

Ventajas de las páginas web SPA

- Desarrollar una página web SPA tiene muchas ventajas frente al MPA. Algunas de ellas son:
- Se tratan de aplicaciones fáciles de desarrollar, desplegar y depurar. Al contar con infinitos frameworks y librerías, será muy sencillo de desarrollar.
- Nos muestran el contenido de forma sencilla y elegante al cargar todo el contenido en una única página.
- Además, ejecutando la lógica del lado del navegador, hace que las interacciones del usuario se ejecuten más rápido.
- La analítica de la web es mucho más sencilla puesto que sólo analiza una única página.
- Muy útiles para la creación de landing page

Desventajas de las páginas web SPA

- El SEO no es tan fácil como en una MPA, aunque esto es cada vez más sencillo de optimizar y no supone tanto problema como hace un tiempo.
- En sitios muy grandes, el mantenimiento del código puede ser algo complicado y hay que tener desde el principio muy clara la estructura de la página.
- Al tener el código fuente directamente en el navegador, si no seguimos buenas prácticas, la seguridad del sitio se podría ver expuesta.
- Aunque las SPA son muy rápidas, al cargar la página por primera vez de forma completa, esto hace que esta carga sea algo lenta, lo que puede ahuyentar posibles clientes. De todos modos, actualmente, hay métodos para mejorar esta velocidad de carga, que veremos en otro post.

Conclusiones sobre las SPA

En conclusión, una MPA consiste en que cada vez que visitemos una página específica de una web se cargará desde cero. Mientras que, en las SPA, al visitar la página por primera vez se cargará completamente su estructura y, cada vez que hagamos click en un enlace interno, sólo solicitaremos al servidor el contenido ya que HTML, CSS y JS están ya cargados. La gran diferencia, entonces, entre una multipage application y una single page application es a la hora de navegar por la página web. Mientras que, con una SPA, por cada enlace dentro de la web solo se cargará el contenido, que es lo que nos envía el servidor vía ajax; con una MPA se cargará todo desde cero.

Modelo Vista Controlador (MVC)

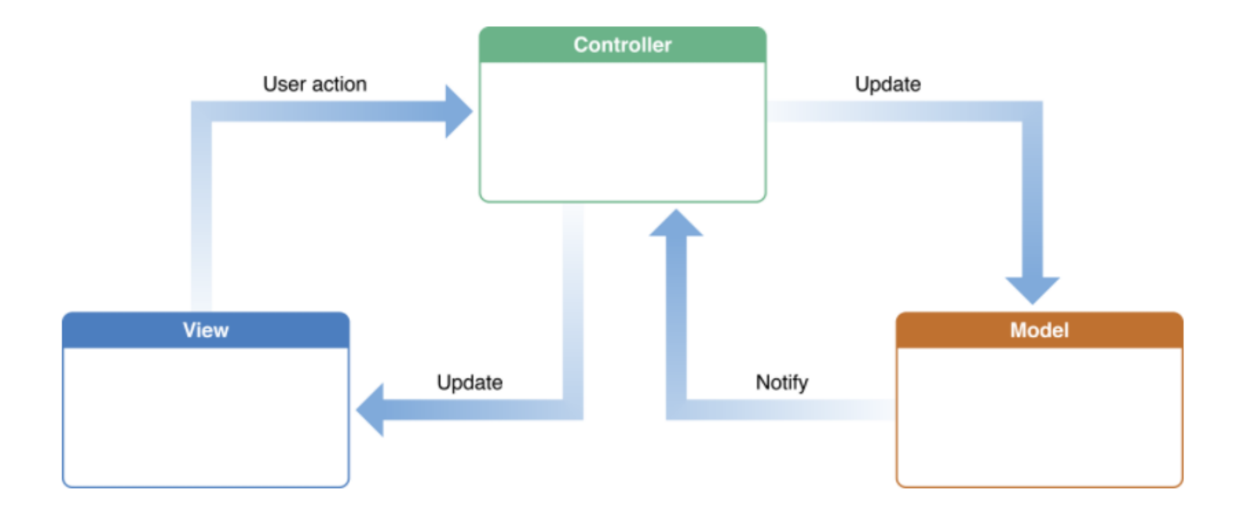
El modo MVC (Model-View-Controller) es un modo de arquitectura de software en ingeniería de software que divide el sistema de software en tres partes básicas: Modelo, Vista y Controlador.

El propósito del modo MVC es lograr un diseño de programa dinámico, simplificar la posterior modificación y expansión del programa y hacer posible la reutilización de una determinada parte del programa. Además, el modo MVC hace que la estructura del programa sea más intuitiva al simplificar la complejidad. Si bien el sistema de software separa sus propias partes básicas, también le da a cada parte básica sus funciones. Los profesionales pueden utilizar su propia experiencia para agrupar grupos relacionados: Modelo: Las funciones que deben tener los programadores al escribir programas (para lograr algoritmos, etc.), y los expertos en bases de datos realizan la gestión de datos y el diseño de bases de datos (que pueden lograr funciones específicas); Controlador: Responsable de reenviar solicitudes y procesar solicitudes; Vista: los diseñadores de interfaces llevan a cabo el diseño de interfaces gráficas.

El patrón de diseño Model-View-Controller (MVC) asigna a los objetos de una aplicación uno de los tres roles: modelo, vista o controlador. El patrón define no solo los roles que desempeñan los objetos en la aplicación, sino que también define la forma en que los objetos se comunican entre sí. Cada uno de los tres tipos de objetos está separado de los demás por límites abstractos y se comunica con objetos de los otros tipos a través de esos límites. La colección de objetos de un determinado tipo MVC en una aplicación a veces se denomina *capa,* por ejemplo, capa de modelo.

MVC es fundamental para un buen diseño para una aplicación. Los beneficios de adoptar este patrón son numerosos. Muchos objetos en estas aplicaciones tienden a ser más reutilizables, y sus interfaces tienden a estar mejor definidas. Las aplicaciones que tienen un diseño MVC también son más fácilmente extensibles que otras aplicaciones. Además, muchas tecnologías y arquitecturas se basan en MVC y requieren que sus objetos personalizados desempeñen uno de los roles de MVC.

La descripción del patrón MVC se muestra en la siguiente figura:



Veamos los detalles de los tres componentes del patrón MVC:

- **Modelo (Model):** se utiliza para encapsular datos relacionados con la lógica empresarial de la aplicación y el método de procesamiento de los datos. Model tiene derecho a acceder directamente a los datos, como el acceso a la base de datos. El modelo no depende de la vista y el controlador, es decir, al modelo no le importa cómo se mostrará u operará. Sin embargo, los cambios en los datos del modelo generalmente se anuncian mediante un mecanismo de actualización. Para implementar este mecanismo, las Vistas utilizadas para monitorear este Modelo deben registrarse en este Modelo con anticipación, de modo que la Vista pueda comprender los cambios que se han producido en el Modelo de datos. (Por ejemplo, el "modo de observador" en el modo de diseño de software)
- **Vista (View):** Capaz de lograr una visualización intencionada de datos (teóricamente, esto no es necesario). Por lo general, no hay lógica de programa en View. Para realizar la función de actualización en la Vista, la Vista necesita acceder al modelo de datos (es decir, el Modelo) que monitorea, por lo que debe registrarse de antemano con los datos monitoreados por ella.
- **Controlador (Controller):** desempeña un papel organizativo entre diferentes niveles, utilizado para controlar el flujo de la aplicación. Procesa eventos y responde. Los "eventos" incluyen el comportamiento del usuario y los cambios en el modelo de datos.

Objetos de modelo

Los objetos de modelo encapsulan los datos específicos de una aplicación y definen la lógica y el cálculo que manipulan y procesan esos datos. Por ejemplo, un objeto modelo puede representar un personaje en un juego o un contacto en una libreta de direcciones. Un objeto de modelo puede tener relaciones de uno y de muchos con otros objetos de modelo, por lo que a veces la capa de modelo

de una aplicación es efectivamente uno o más gráficos de objetos. Gran parte de los datos que forman parte del estado persistente de la aplicación (ya sea que ese estado persistente se almacene en archivos o bases de datos) deben residir en los objetos del modelo después de cargar los datos en la aplicación. Dado que los objetos modelo representan el conocimiento y la experiencia relacionados con un dominio de problema específico, se pueden reutilizar en dominios de problemas similares. Idealmente, un objeto de modelo no debe tener una conexión explícita con los objetos de vista que presentan sus datos y permitir a los usuarios editar esos datos; no debe preocuparse por problemas de interfaz de usuario y presentación.

Un objeto modelo es un tipo de objeto que contiene los datos de una aplicación, proporciona acceso a esos datos e implementa lógica para manipular los datos. Los objetos de modelo desempeñan uno de los tres roles definidos por el patrón de diseño Model-View-Controller. (Los otros dos roles son desempeñados por objetos de vista y controlador). Cualquier dato que forme parte del estado persistente de la aplicación (ya sea que ese estado persistente se almacene en archivos o bases de datos) debe residir en los objetos del modelo después de cargar los datos en la aplicación.

Dado que los objetos modelo representan el conocimiento y la experiencia relacionados con un dominio de problema específico, se pueden reutilizar cuando ese dominio de problema está en vigor. Idealmente, un objeto de modelo no debe tener una conexión explícita con los objetos de vista que presentan sus datos y permitir a los usuarios editar esos datos, en otras palabras, no debe preocuparse por problemas de interfaz de usuario y presentación

Comunicación: Las acciones del usuario en la capa de vista que crean o modifican datos se comunican a través de un objeto controlador y dan como resultado la creación o actualización de un objeto modelo. Cuando un objeto de modelo cambia (por ejemplo, se reciben nuevos datos a través de una conexión de red), notifica a un objeto controlador, que actualiza los objetos de vista adecuados.

Objetos Vista

Un objeto de vista es un objeto de una aplicación que los usuarios pueden ver. Un objeto de vista sabe cómo dibujarse a sí mismo y puede responder a las acciones del usuario. Un propósito principal de los objetos de vista es mostrar datos de los objetos de modelo de la aplicación y permitir la edición de esos datos. A pesar de esto, los objetos de vista suelen desacoplarse de los objetos de modelo en una aplicación MVC.

Comunicación: Los objetos de visualización aprenden sobre los cambios en los datos del modelo a través de los objetos del controlador de la aplicación y comunican los cambios iniciados por el usuario (por ejemplo, el texto introducido

en un campo de texto) a través de los objetos del controlador a los objetos del modelo de una aplicación.

Objetos Controlador

Un objeto controlador actúa como intermediario entre uno o más de los objetos de vista de una aplicación y uno o más de sus objetos de modelo. Los objetos controlador son, por lo tanto, un conducto a través del cual los objetos de vista aprenden sobre los cambios en los objetos del modelo y viceversa. Los objetos de controlador también pueden realizar tareas de configuración y coordinación para una aplicación y administrar los ciclos de vida de otros objetos.

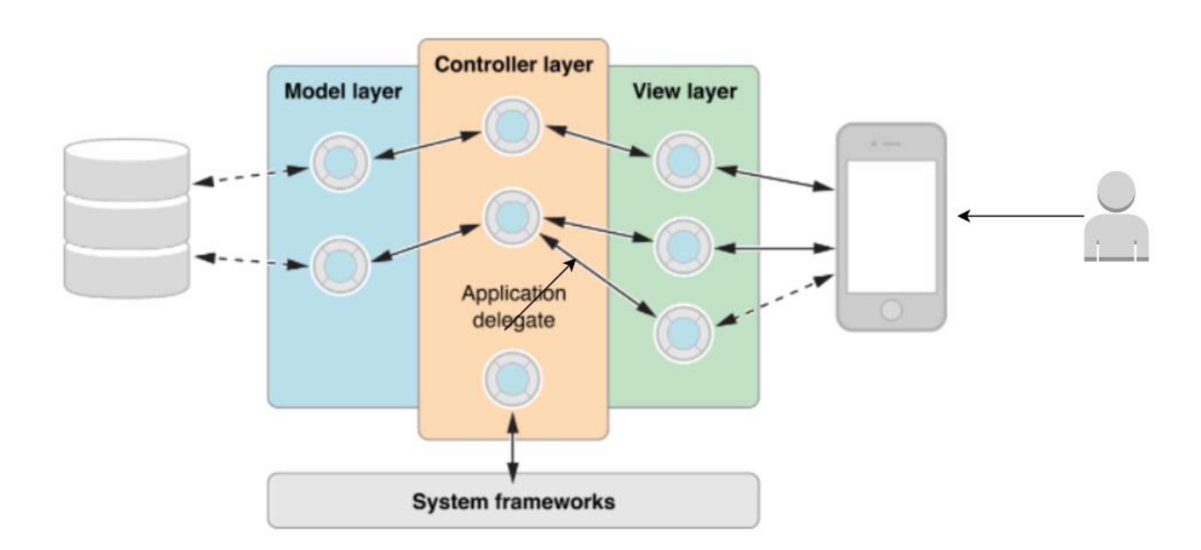
El controlador actúa como coordinador o como intermediario entre uno o más objetos de vista y uno o más objetos de modelo. En el patrón de diseño Model-View-Controller, un objeto controller (o, simplemente, un *controller)* interpreta las acciones e intenciones del usuario realizadas en los objetos de vista, como cuando el usuario pulsa o hace clic en un botón o introduce texto en un campo de texto, y comunica datos nuevos o modificados a los objetos del modelo. Cuando los objetos del modelo cambian (por ejemplo, el usuario abre un documento almacenado en el sistema de archivos), comunica esos nuevos datos del modelo a los objetos de vista para que puedan mostrarlos. Los controladores son, por lo tanto, el conducto a través del cual los objetos de vista aprenden sobre los cambios en los objetos del modelo y viceversa. Los objetos de controlador también pueden configurar y coordinar tareas para una aplicación y administrar los ciclos de vida de otros objetos

Coordinación de controladores: Los controladores coordinadores supervisan y administran el funcionamiento de toda una aplicación, o parte de una. A menudo son los lugares donde se inyecta lógica específica de la aplicación en la aplicación. Un controlador de coordinación cumple una variedad de funciones, que incluyen:

- Responder a mensajes de delegación y observar notificaciones
- Responder a mensajes de acción (que son enviados por controles como botones cuando los usuarios los pulsan o hacen clic en ellos)
- Establecer conexiones entre objetos y realizar otras tareas de configuración, como cuando se inicia la aplicación
- Gestión del ciclo de vida de los objetos "propios

Comunicación: Un objeto controlador interpreta las acciones del usuario realizadas en los objetos de vista y comunica datos nuevos o modificados a la capa del modelo. Cuando los objetos del modelo cambian, un objeto controlador comunica esos nuevos datos del modelo a los objetos de vista para que puedan mostrarlos.

Desde la comprensión general del patrón MVC, existe una conexión directa entre la capa de vista y la capa del modelo, y los cambios en la capa del modelo se reflejarán a través de la capa de vista



¿Por qué Angular?

Repasemos qué es Angular y por qué elegimos esta tecnología para el desarrollo frontend, luego veamos cómo instalarlo para empezar a trabajar en nuestros propios proyectos. Será necesario que analicemos cómo es la estructura de los proyectos creados con esta tecnología y luego empezaremos a escribir nuestras primeras líneas de código.

¿Qué es Angular?

Angular es un framework open source desarrollado por Google para facilitar la creación y programación de aplicaciones web de una sola página o SPA (Single Page Application).

Angular separa el frontend y el backend en la aplicación, evita escribir código repetitivo y mantiene todo más ordenado gracias a su patrón MVC (Modelo-Vista-Controlador) asegurando los desarrollos con rapidez, a la vez que posibilita modificaciones y actualizaciones.

Ventajas

Ahorras tiempo: Cuando empezamos a pensar en cómo crear una aplicación web, tenemos que tomar una serie de decisiones; por ejemplo, la arquitectura de la aplicación, su organización, etc. Angular ya lo hace por nosotros, al ser un framework nos establece una metodología de trabajo,

haciendo que nos enfoquemos en cosas más importantes, como las funcionalidades de la aplicación web.

CLI integrado: Permite escribir comandos para crear archivos, módulos, componentes, servicios, clases, etc, de igual forma crear versiones para publicar en ambiente de desarrollo, pruebas, producción y aplicar pruebas unitarias.

Usa lenguaje TypeScript: Al usar un lenguaje fuertemente tipado nos obliga a ser consistentes en el desarrollo.

Gran documentación y comunidad: Además del respaldo de una gran empresa como Google, Angular goza de tener una gran comunidad de desarrolladores y muchos ejemplos y tutoriales.

Fáciles de mantener: Al usar TypeScript, cualquier cambio que deba hacerse en la aplicación podrá llevarse a cabo rápidamente y con menor probabilidad de errores. Hace posible intercambiar o añadir programadores en los proyectos. Cualquier programador de Angular podrá leer el código escrito por otro programador de Angular y comprenderlo muy rápidamente. Esto es una gran ventaja a la hora de trabajar en equipo

El futuro es estable: Luego de tantos años de desarrollo, y su gran comunidad, tomar Angular como entorno de desarrollo nos da la tranquilidad de que seguirá evolucionando e integrando nuevas funcionalidades sin comprometer el trabajo realizado.

Databinding que es la forma que tiene Angular para permitirnos mostrar contenido dinámico

Expresiones HTML que permiten extender la funcionalidad del código desarrollando encima del código HTML, pero teniendo acceso a otros componentes útiles que agregan valor semántico a nuestras aplicaciones enriqueciendo el HTML, por medio de lo que se conoce como "directiva".

Inyección de dependencias: A través de Angular se puede hacer uso de la inyección de dependencia para tener diferentes funcionalidades como podría ser un servicio que haga consultas a una API y tenerlo disponible para usarlo en cualquier parte de nuestro proyecto solo con especificarlo en el constructor. De esta manera se crea un sistema más modular.

Componentes: Los componentes son una característica muy poderosa en Angular que permite desarrollar porciones de código HTML con código TypeScript para que tengan una funcionalidad específica. Por ejemplo, para listas datos, mostrar el detalle de un item, presentar una ventana emergente (pop up) o llenar una cuadrícula.´

En adelante cualquier duda o consulta te sugerimos que revises la documentación oficial de Angular y su versión en español. El equipo de desarrollo oficial sugiere en su sitio algunos recursos para complementar

Instalación de Angular

No hay un requerimiento especial para la instalación, podemos usar cualquier sistema operativo que nos guste y un mínimo de espacio en el disco. Los pasos a seguir requieren que tengamos abierta nuestra consola de comandos, acceso a internet y un IDE o entorno de desarrollo que nos guste como Visual Studio Code o SublimeText. ¡Comencemos!

1. Instalar NodeJS en su última versión desde el sitio de web oficial
2. Abrimos la consola del sistema operativo
    - Windows: puedes verlo haciendo clic aquí
    - Linux: Puedes verlo haciendo clic aquí
    - MAC OS: Puedes verlo haciendo clic aquí

1. En la consola ejecutar el siguiente comando: *npm install -g npm@latest*

¿Qué es npm?

npm es el Manejador de Paquetes de Node.js (Node Package Manager) que viene incluido y ayuda a cada desarrollo asociado a Node. Es ampliamente utilizado por los desarrolladores de JavaScript para compartir herramientas, instalar módulos y administrar dependencias.

4. Instalar la última versión de Angular CLI con el siguiente comando: ***npm install -g @angular/cli@latest***

Ya puedes crear un proyecto con angular utilizando ese comando.

5. Moverse desde la consola hasta el lugar donde queremos crear nuestro proyecto. Mi escritorio, mis documentos, o cualquier otra carpeta específica.
6. Ejecutar el siguiente comando: ***ng new nombre-del-proyecto***
7. El asistente nos solicitará algunos datos del proyecto:

- El nombre del proyecto no puede contener espacios o guiones u otros caracteres especiales, solo letras y números.
  Si se quiere añadir el routing de angular. Esto es opcional, ya que se puede agregar de forma manual de ser necesario.
  Preguntará si queremos usar un formato específico para los estilos.

8. Esperamos a que el asistente acabe de generar nuestro proyecto de Angular

9. Ya estamos listos para abrir el proyecto con nuestro IDE

Estructura de un proyecto Angular

Al crear un proyecto en Angular, existe una estructura predefinida que debemos respetar para mantener el orden del proyecto. Existen varias formas de organizar nuestros archivos del proyecto, y con algunas diferencias entre los desarrolladores, hay un consenso general en cómo se deben definir algunos directorios y archivos. En la siguiente imagen se puede ver cómo se organizan las carpetas y archivos:







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

```
Prueba
  e2e
  node_modules
  src
    app
      services
        login.service.ts
      shared
        form
        table
      vista1
      vista2
      app-routing.module.ts
      app.component.html
      app.component.scss
      app.component.ts
      app.component.spec.ts
      app.module.ts
    assets
    environments
    styles
      buttons.scss
      table.scss
    favicon.ico
    index.html
    main.ts
    polyfills.ts
    styles.scss
    test.ts
  .editorconfig
  .gitignore
  angular.json
```

Al crear un nuevo proyecto las tres principales carpetas son:

- **e2e**: Carpeta dedicada a testing “end to end”.
- **node_modules**: Carpeta en la que se incluyen todos los node modules instalados al hacer npm install.
- **src**: Carpeta en la que se realizará el desarrollo de la aplicación.

Es importante mencionar que la carpeta node_modules, en general se agrega al .gitignore para no enviar al repositorio los archivos de paquete que pueden ocupar una cantidad importante de espacio.

Luego hay varios archivos con diferentes extensiones, en su mayoría son archivos de configuración del proyecto. Para no complicar demasiado en este punto del capítulo, sólo miraremos el package.json

package.json

Este es uno de los archivos principales del proyecto. Tal y como indica la extensión del archivo es un JSON y dentro de él se encuentra la información relevante sobre el proyecto:

- **name**: Indica el nombre del proyecto.
- **version**: Indica el número de versión del desarrollo. A medida que se vayan desplegando diferentes versiones, se deberá ir incrementando este número.
- **scripts**: Son los comandos que permite ejecutar la consola. Se podrán ejecutar estos comandos de 2 formas diferentes:
    1. Poniendo en la consola “npm run [clave] “. Por ejemplo, para ejecutar el comando llamado “start”, comando que arranca el proyecto, o podemos escribir “npm run start”.
    2. Poniendo en la consola “ng” seguido de uno de los valores que se muestran en el JSON. Siguiendo con el mismo ejemplo anterior, para ejecutar el comando “start” se escribe en este caso “ng serve”.
- **dependencies**: Indica las dependencias del proyecto (para la versión de producción) como pueden ser la versión del compilador de Angular, versión del router de Angular, versión de RxJS, Axios, etc. Si se añaden nuevas librerías / dependencias mediante npm install aparecerán en este apartado.
- **devDependencies**: Aquí se indican las dependencias que son necesarias durante el desarrollo del proyecto como pueden ser la versión del Angular CLI, versión de TypeScript o la versión de librerías para testing como pueden ser Jasmine y Karma. Si queremos que al instalar una librería sea indicada como una dependencia de desarrollo hay que agregar ***--save-dev*** el comando quedaría: ***npm install <nombre-del-paquete> --save-dev***

Carpeta src

En esta carpeta es donde se encuentra todo el código del proyecto (componentes, estilos, configuraciones de entornos, archivos de traducciones, etc.) Los elementos

más relevantes son app, assets, environments y el archivo index.xml, que es el html principal, a partir del cual se cargará el resto de la aplicación.

Carpeta app

En esta carpeta se encuentran los componentes, módulos y archivos de rutas. Se deberán crear tantas subcarpetas como módulos y/o vistas tenga la aplicación

Componentes: Son los diferentes elementos que van a componer la aplicación, por ejemplo, una tabla, una ventana modal o un formulario. Los componentes están formados por 3 tipos de archivos principales: un archivo HTML, un archivo CSS y un archivo TypeScript. En el HTML se define cómo se verá el componente. En el archivo de TypeScript el comportamiento o lógica del componente. En el CSS se definen los estilos propios del componente.

Módulos: En estos módulos (archivos “module.ts”) se importan los componentes que pertenecen a dicho módulo. Permiten organizar la aplicación y reutilizar componentes pertenecientes a otros módulos. Puede crearse un módulo por cada vista de la aplicación y adicionalmente se puede crear un módulo que englobe a los componentes comunes o compartidos por varias vistas. Además, disponer de diferentes módulos permite beneficiar la carga perezosa (lazy loading) que aporta un mayor rendimiento en la aplicación debido a que sólo se cargarán aquellos módulos que sean necesarios en cada momento. Al crear una aplicación vendrá un único módulo por defecto: “app.module.ts”.

Archivos de rutas: Nos permiten indicar las rutas que tendrán las diferentes vistas de la aplicación. Las rutas se configuran dentro de los archivos llamados routing.module.ts, puede haber rutas principales y también rutas anidadas.

Carpeta assets

Esta carpeta está dedicada a guardar los diferentes recursos que necesite una aplicación como imágenes, iconos, tipos de fuentes o archivos de traducciones si la aplicación requiere utilizar varios idiomas.

Un comentario final:

Para practicar, o si necesitáramos tener un proyecto de forma rápida Stackblitz es un IDE para angular que funciona en una página web y que te permite crear aplicaciones de Angular desde el navegador. ¡Eso quiere decir que no tienes que instalar nada más!

Arquitectura Angular

Ya sabemos qué es Angular y por qué lo elegimos. Ahora vamos a meternos en sus componentes, entender las piezas que lo forman y cómo se encastran para permitirnos crear aplicaciones webs de forma rápida, eficiente y escalables.

Angular es un framework grande y complejo por lo cual siempre será recomendable que acompañes la lectura apoyándote en la documentación oficial para obtener mayores detalles y buscar en google otros ejemplos que te permitan tener una mayor perspectiva de los conceptos.

Recorda que la mejor forma de aprender es practicando. Si leer es el primer paso, codear es el segundo.

Arquitectura

La arquitectura de una aplicación en Angular se basa en bloques de construcción, llamados Módulos o NgModules y proporcionan un contexto de compilación para cada uno de los componentes que hacen a la página. Cualquier sitio web creado con Angular constituye un conjunto de módulos y componentes.

Hay más elementos constructivos en Angular y se pueden clasificar en tres bloques principales:

- **Módulos**: NgModule
Componentes: @Component, @Template, @Directives, Data-binding, etc
Servicios e Inyección de dependencias: @Injectable, Routing, etc

A parte, están los modelos de datos que se designan a través de clases o interfaces con TypeScript que luego se importan al momento de usarlos.

Módulos en Angular

Las aplicaciones de Angular son modulares, es decir, se componen de varios bloques independientes, los cuales cada uno contiene una parte de la aplicación o una serie de comportamientos de esta. Los módulos son la manera básica de organizar una aplicación en este framework. En un módulo estarán una serie de archivos que están relacionados entre sí y organizados en una misma carpeta o directorio. Esto permite una estupenda organización del código y sobre todo permite trabajar con muchos otros elementos externos que brindarán una estupenda funcionalidad a la aplicación.

Crear módulos

Para crear un módulo propio, además del módulo principal que se genera al crear un proyecto (app.module.ts) se deben seguir los siguientes pasos:

1. Ir a la consola o línea de comandos desde la terminal que resulte más cómoda
2. Ejecutar el comando: ng generate module <nombre-modulo> o su abreviado: ng g m <nombre-modulo>

Una vez ejecutado el comando, AngularCLI creará un subdirectorio con el nombre del módulo dentro de la carpeta "src/app" y un archivo: nombre-modulo.module.ts.

Angular CLI aplica las convenciones de nombres más adecuadas y como los módulos son clases, internamente les coloca en el código la primera letra siempre en mayúscula.

Anatomía de un módulo

Cuando creamos un módulo, los primero que veremos es algo como la siguiente imagen:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class LayoutModule { }
```

Como se puede observar, Angular define los módulos como clases, a través del decorador @NgModule. En el mismo, contiene dos arrays bien definidos:

- **imports**: Clases exportadas importaciones necesarias. Le dice a Angular sobre otros NgModules que este módulo en particular necesita para funcionar correctamente (https://docs.angular.lat/guide/bootstrapping)
- **declarations**: Aquí se listan los componentes u otros artefactos que incluye este módulo.

Pudiendo además, agregarse lo siguientes:

- **providers:** Enumera los proveedores de servicios necesarios.
- **bootstrap:** El componente raíz que Angular crea e inserta en la página web de host index.html (https://docs.angular.lat/guide/bootstrapping)
- **exports**: Aquí se listan los componentes exportados hacia afuera del módulo.
- En el siguiente ejemplo, inicia el módulo con AppComponente y se declara el uso de otro módulo más, LoginComponent. Además este módulo requiere del uso de otros 3 módulos para funcionar:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { LoginComponent } from './login/login.component';


@NgModule({
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule ],
  declarations: [ AppComponent, LoginComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

¿Quién inicia el módulo?

La aplicación arranca en el main.ts donde se le pasa el primer módulo al sistema para que luego se propague la ejecución cargando los demás módulos.

Componentes en Angular

Componentes

Tomando los conocimientos ya vistos de MVC, entendemos que el proyecto tiene separadas las acciones correspondientes a la lógica, el modelo de datos y las vistas. Una aplicación de Angular tiene una serie de componentes que se comunican con elementos externos mediante servicios.

Un componente en Angular es un elemento que está compuesto por:

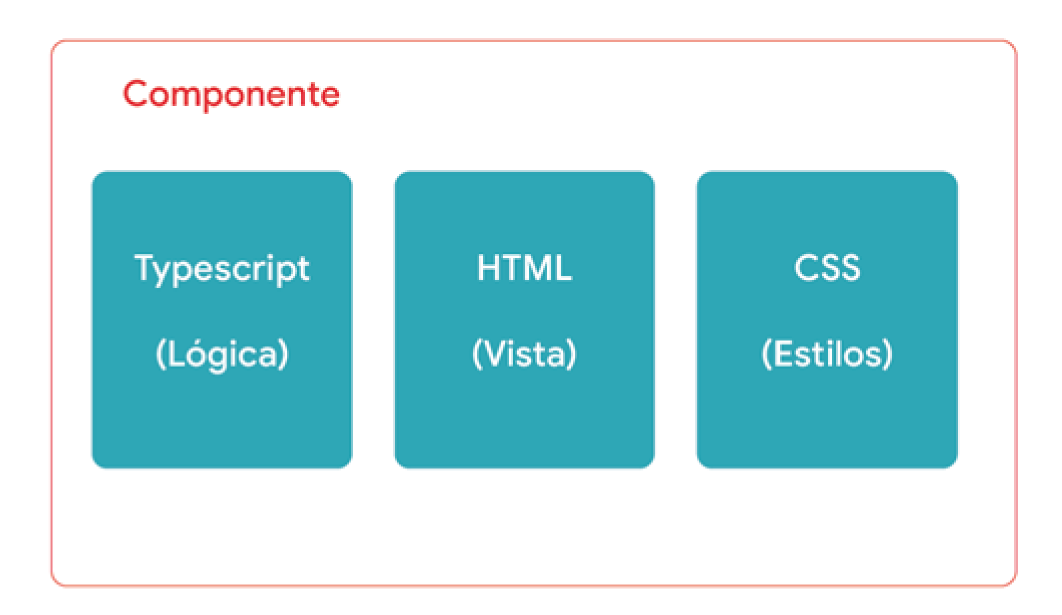
- **Un Template** (app.component.html), que contendrá el HTML para visualizar la interfaz de usuario, la vista o en términos más simples lo que vas a ver en la página.

- **Un Controlador o Controller** que es donde se encuentra la lógica, (app.component.ts), ese archivo debe incluir una clase y esta es la que va a contener las propiedades que se van a usar en la vista (HTML) y los métodos que será las acciones que se ejecutarán en la vista. En este archivo de lógica también se incluye una metadata, que es definida con un decorador, que identifica a Angular como un componente.

- **Un archivo para el CSS** (podemos usar un preprocesador como SASS o LESS), donde incluiremos los estilos, lo que nos ayuda a hacer bonita nuestra aplicación.

Un componente puede tener más archivos, dependiendo de las necesidades de los proyectos. Al final de la cuenta, una aplicación en Angular estará compuesta por varios componentes.

Cómo vemos en la siguiente imagen, un componente se forma de una lógica compuesta por archivos de TypeScript (.ts), las vistas o templates compuestos por archivos HTML y archivos de estilo compuestos por CSS:



Hay otros archivos adicionales que pueden formar parte de un componente como los archivos de testing que veremos mas adelante.

Es importante que empecemos a ver las aplicaciones webs como un conjunto de componentes donde cada bloque cumple una función bien definida. A su vez cada componente tendrá padre, hijos y/o hermanos. Si hacemos una buena planificación de los componentes, podremos abstraernos lo suficiente y luego poder reutilizarlos.

En la siguiente imagen podemos apreciar como el SotyLineComponente tiene un componente hijo que se repite StoryItemComponent y cada repetición de este componente es una instancia particular. ¿Empezamos a ver las piezas del rompecabezas?:

HeaderComponent
StoryLineComponent
StoryItemComponent
CardComponent
FooterBtnComponent
FooterComponent

[Componentes Angular.](#) *(n.d.).*

Repasemos la imagen anterior:

HeaderComponent	ComponenteEncabezado: Este componente ancapsula subcomponentes referidos a la función de encabezado con una alineación en fila.
StoryLineComponent	ComponenteLineaHistorias: Es un componente padre, con una alineación de fila para sus componentes hijos.
StoryItemComponent	ComponenteItmHistorias: Es un componente hijo que se instancia para cada historia. Cada instancia tendrá sus propiedades particulares como una imagen y una etiqueta. Todas sus instancias son encapsuladas dentro del componente padre ComponenteLineaHistorias.
CardComponent	ComponenteTarjeta: Este componente es una alineación vertical o de columna donde se encapsulan todos los componentes referidos a las tarjetas.







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

FooterComponent	ComponentePie: Es un componente padre con una alineación en fila para sus componentes hijos.
FooterBtnComponent	ComponenteBotonPie: Cada botón es una instancia con sus propias características funcionales además de un icono en particular. Todas las instancias son contenidas dentro del componente padre ComponentePie.

Ya vimos como crear un proyecto en Angular usando el CLI pero repetimos el comando:

ng new nombre-de-mi-app

Ahora, para crear un componente se debe ejecutar el siguiente comando:

ng generate component <nombre-del-componente> [opciones]

También se puede usar la forma abreviada:

ng g c <nombre-del-componente> [opciones]

Y para un mejor orden se puede especificar una carpeta donde ir guardando los componentes, por ejemplo en el siguiente comando, incorporamos la carpeta components:

ng g c components/<nombre-del-componente> [opciones]

El componente recién creado tendrá una forma parecida a la siguiente:

```
// Importamos el core de componentes de Angular
import { Component } from '@angular/core';

// ¡El Component es un decorador!
@Component({
  // selector: Definimos el nombre con el cual se va a llamar o montar al camponente
  // <app-root></app-root>
  selector: 'app-root',
  // templateUrl: Definimos el archivo HTML del template del componente
  templateUrl: './app.component.html',
  // styleUrls: cargamos los archivos de stilo en un array
  styleUrls: ['./app.component.css']
})

// Exportamos la clase para ser usada dentro de nuestro proyecto
export class AppComponent {
  title = 'app';
}
```







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

De la imagen anterior podemos inferir en 3 partes importantes que hacen a un componente en Angular y que se encuentran dentro del decorador @Component():

1. **selector,** le dice a Angular que cree e inserte una instancia de este componente siempre que encuentre la etiqueta en el html. Por ejemplo, si el HTML de una aplicación contiene <app-root></app-root>, entonces Angular inserta una instancia de la vista HeaderComponent entre esas etiquetas.
2. **templateUrl,** la dirección relativa al template HTML del componente. Alternativamente, se puede escribir código html.
3. **stylesUrl,** la dirección relativa al archivo CSS del componente.

En pocas palabras, el selector es el nombre con el cual luego invocamos al componente en el html como se puede observar a continuación:

https://argentinaprograma-main.stackblitz.io

Usando componentes

Al crear una aplicación en angular, trabajamos con una jerarquía de módulos y componentes por lo que es importante comprender cómo manipularlos.

Exportar Componentes del módulo hacia afuera

Por defecto los componentes definidos dentro de un módulo sólo son accesibles por éste. Si deseamos dejar visibles componentes, para que luego sean utilizados desde otros componentes, simplemente deberemos invocarlos en el array exports del módulo como sigue:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeaderComponent } from './header/header.component';
import { NavComponent } from './nav/nav.component';
import { FooterComponent } from './footer/footer.component';

@NgModule({
  declarations: [
    HeaderComponent,
    NavComponent,
    FooterComponent,
  ],
  imports: [
    CommonModule
  ],
  // Exportamos el componente para ser usado por otros modulos
  exports: [
    HeaderComponent,NavComponent, FooterComponent,
  ]
})
export class LayoutModule { }
```

Importar componentes de otros módulos

Para hacer uso de los componentes declarados en otro módulo, se debe importar el módulo completo que contiene dichos componentes al módulo destino. Para ello, debemos editar el módulo destino:

1. Agregar la referencia relativa del módulo.
2. Referenciar en el array imports como en el siguiente ejemplo:







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { LayoutModule } from './layout/layout.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    // Importo el LayoutModule desde otro modulo
    LayoutModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Templates, Plantillas o Vistas en Angular

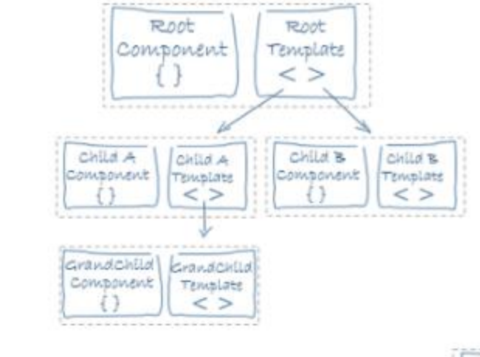
Ahora los componentes requieren de un template o plantilla HTML para que las vistas puedan ser interpretadas por el navegador web. Un template es un bloque HTML que le dice al framework, en este caso Angular, cómo renderizar o dibujar el componente definiendo así la vista con la cual interactúa el usuario. El template es un código HTML dotado de la capacidad de interpretar o agregar funciones, variables, lógica y bucles (ifs y fors), el sistema de binding que veremos más adelante y que permite enlazar las variables al código HTML. Todas estas capacidades extras del template son gracias a las expresiones y directivas que veremos en la siguiente sección.

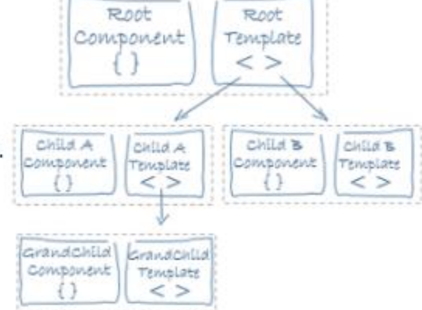


Angular Template. (n.d.). [Ilustator]. Angular.



Como se dijo antes, si tenemos una jerarquía de componentes, también podemos pensar que existe una jerarquía de vistas, las cuales contienen vistas embebidas o anidadas, o incluso dentro de otros componentes. Es importante empezar a pensar como un proyecto es un árbol de componentes y vistas que se relacionan entre sí. Las vistas, dotadas de la lógica, permiten modificar, mostrar y ocultar secciones enteras de la UI o páginas como una unidad.





Árbol de componentes. (n.d.). [Ilustrator]. Árbol de Componentes.

Veamos un ejemplo de un template de Angular donde lo importante es observar 2 detalles. El primero es que se usan además de HTML otras expresiones que veremos más adelante y lo segundo, e importante para este momento del capítulo, es la posibilidad de introducir un template a dentro de otro.

```
<h2>Hero List</h2>

<p><i>Select a hero from the list to see details.</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-
detail>
```

Este fragmento de código es de un template llamado hero-list.component.html, y que pertenece a un controlador hero-list.controller.ts. Dentro de este template, tenemos referenciado otro template app-hero-detial al cual se le pasan ciertos datos a través de atributos al estilo HTML.

Expresiones y Pipes en Angular

Expresiones

Con las expresiones se enriquece el HTML, ya que permite interpretar expresiones y resolverlas dentro de los templates. Las expresiones de Angular son muy similares a las expresiones de JavaScript. Las mismas pueden contener literales, operadores, variables y funciones. Para definir una expresión simplemente se las engloba dentro de dobles llaves {{}} que se podrán colocar expresiones en cualquier lugar del HTML. Cuando se ejecute la aplicación, las expresiones se evaluarán, resolverán y serán sustituidas con el resultado que corresponda. Veamos algunos ejemplos de expresiones.

- <h1>{{ 1 + 1}}</h1> esto se renderiza como <h1>2</h1>
- {{ "Hola " + "JavaScript" }} esto se renderiza como Hola JavaScript

Otra aplicación interesante de las expresiones es la capacidad de formatear la salida, por ejemplo, diciendo que lo que se va a escribir es un número y que deben representarse dos decimales necesariamente. Por ejemplo:

{{ precio | number:'2'}}

Estos formateadores son conocidos como Pipes y Angular ya nos provee un grupo de funciones que podemos usar:

- **DatePipe:** Cambia el valor de fecha.
- **UpperCasePipe:** Transforma texto en Mayúscula.
- **LowerCasePipe:** Transforma el texto en minúscula.

- **CurrencyPipe:** Transforma un número en una cadena de moneda de acuerdo a ciertas reglas.
- **DecimalPipe:** Transforma un número en una cadena con punto decimal.
- **PercentPipe:** Transforma un número en una cadena de porcentaje.

Además, y muy importante, las expresiones se utilizan para colocar datos de forma dinámica gracias al biding que veremos más adelante y el resultado será, por ejemplo que en el siguiente código HTML, cuando el usuario vaya ingresando dígitos en el input automáticamente aparecerán dentro de la etiqueta span. Esto es gracias al binding. Veamos un ejemplo de cómo quedaría:

```
<input placeholder="ingrese su telefono" type="number" [ngModel]="telefono" />
<p>El número de teléfono es: {{telefono}}</p>
```

Las expresiones no están pensadas para desarrollar la lógica de una aplicación. Sin embargo es posible llamar a funciones definidas en el controlador.

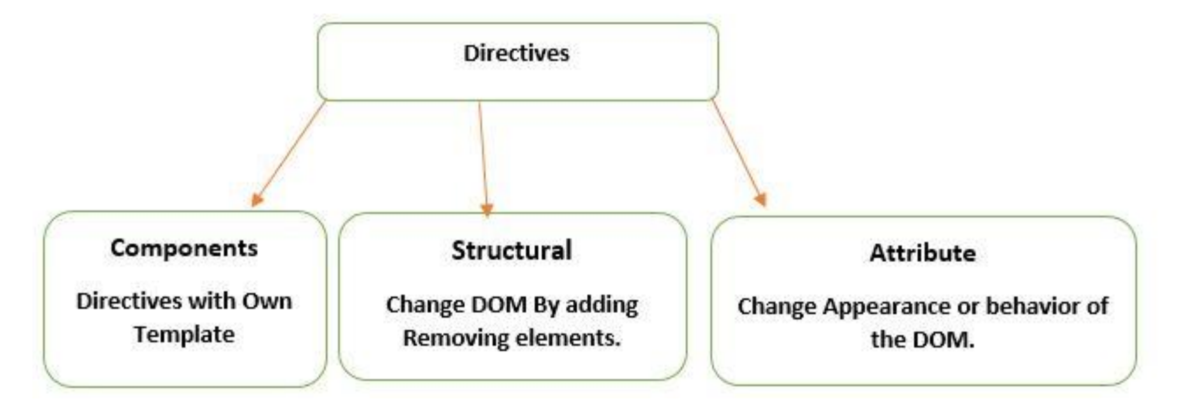
Directivas en Angular

Introducción a las directivas de Angular

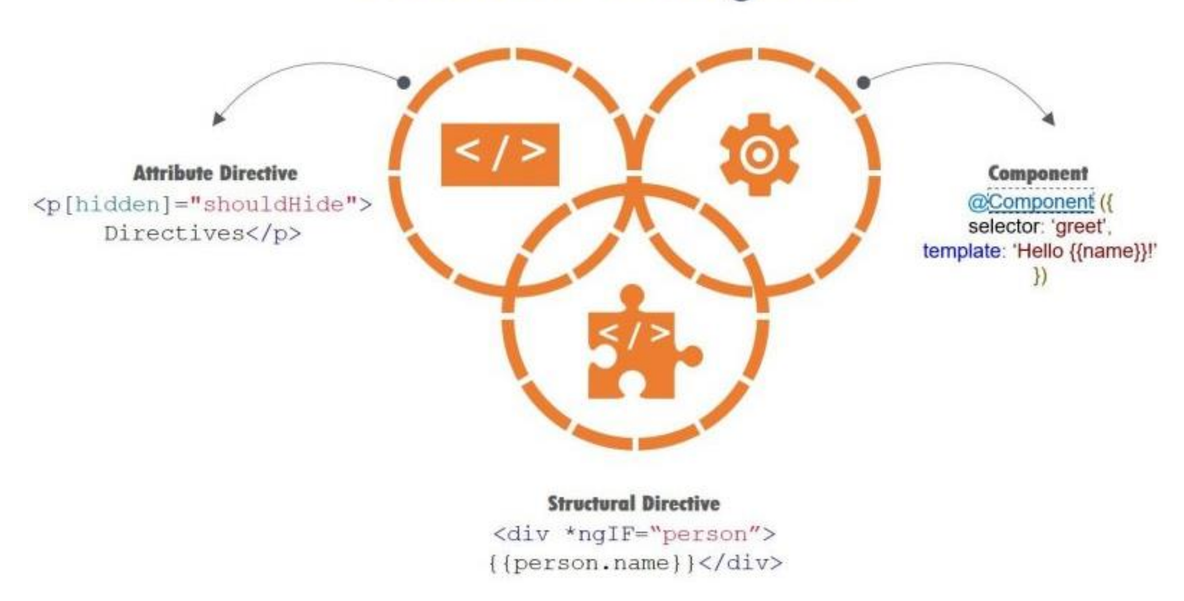
Los templates de Angular son fragmentos de HTML dinámicos, y cuando Angular los renderiza, transforma el DOM de acuerdo con las instrucciones dadas por las directivas. En el fondo de la cuestión, una directiva es una clase con un decorador @Directive() y por lo tanto un componente es técnicamente una directiva. Sin embargo, los componentes son tan distintivos y fundamentales para las aplicaciones de Angular que se define específicamente el decorador @Component(), que extiende el decorador @Directive() con características orientadas a los templates. Además de los componentes, existen otros dos tipos de directivas: estructural y atributo. Angular define una serie de directivas de ambos tipos, y puedes definir las tuyas propias usando el decorador @Directive().

Las directivas aparecen en los templates dentro de una etiqueta de elemento como atributos, ya sea por nombre o como el destino de una asignación o un enlace.

En los siguientes diagramas podemos ver las diferentes funciones que puede tomar una directiva:



Directivas de angular. (n.d.).



Tipos de directivas Angular. (n.d.).

Directivas de Atributo

Alteran la apariencia o comportamiento de un elemento del DOM. Son usados como atributos de los elementos. Se aplican como atributos a los elementos HTML. Modifican el DOM pero sin ser capaces de crear o remover elementos HTML sobre el que se aplican. Las directivas de tipo atributo están formadas por:

- **ngModel**: Implementa binding, que lo veremos en la siguiente sección
- **ngClass**: permite añadir/eliminar varias clases
- **ngStyle**: permite asignar estilos inline

Directivas Estructurales

Permiten añadir, manipular o eliminar elementos del DOM. Se mostrarán algunos ejemplos. Para la lista completa consultar la documentación oficial.

***ngIf o [ngIf]:** Permite incluir condicionales de lógica del HTML, como por ejemplo evaluar sentencias, hacer comparaciones, mostrar u ocultar secciones de código, y entre las muchas condiciones que se pueden crear, para que se renderiza el HTML, cumpliendo la sentencia a evaluar.

En el siguiente ejemplo vemos cómo la directiva ngIf se usa para mostrar el nombre del Heroe si esta propiedad name está definida en el objeto selectHero:

```
<section class="img-section" *ngIf="selectedHero.name">
    <h2>{{selectedHero.name}}</h2>
    <img [src]="selectedHero.img" [alt]="selectedHero.name"/>
</section>
```

***ngFor o [ngFor]:** Permite ejecutar bucles sobre elementos iterativos

En el siguiente ejemplo vemos cómo se repite la lista con el id del heroe y su nombre haciendo un bucle sobre una lista llamada superHeroes:

```
<ul class="heroes">
  <li *ngFor="let hero of superHeroes" (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
```

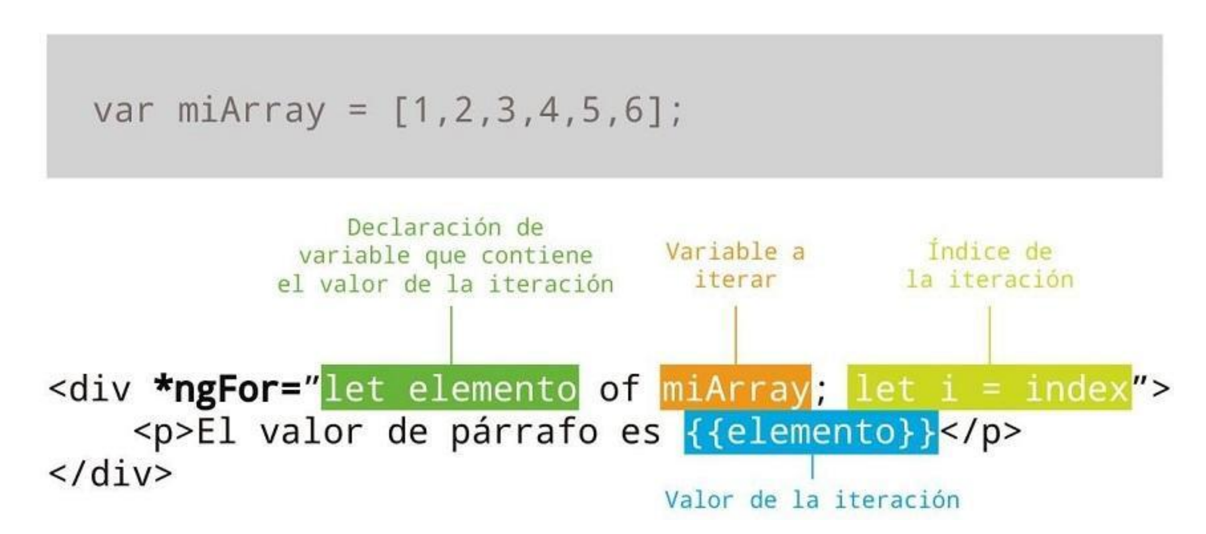
Ahora veamos en la siguiente imagen cómo se compone el ***ngFor:**







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein



***ngSwitch o [*ngSwitch] + *ngSwitchCase o [ngSwitchCase]:** Esta directiva corresponde a una serie de directivas que cooperan entre sí para generar un resultado. Estas directivas son ngSwitch, ngSwitchCase y ngSwitchDefault. La directiva ngSwitch es una directiva de atributo, mientras que las directivas ngSwitchCase y ngSwitchDefault corresponden a directivas estructurales.

Veamos en la siguiente imagen cómo se compone el ngSwitch junto con el ngSwitchCase:



Directivas Estructurales. *(n.d.).*

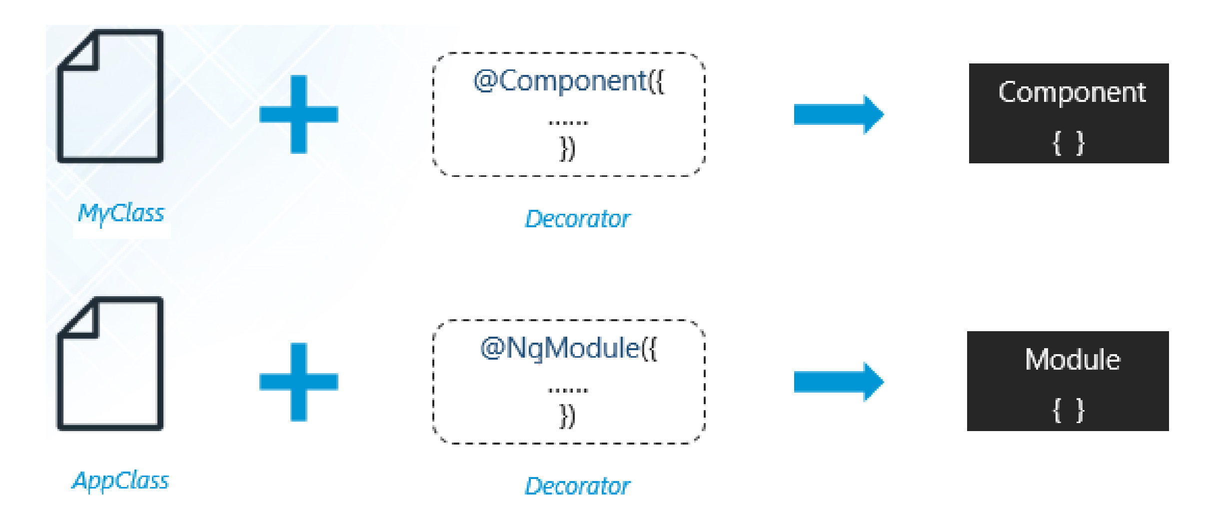
Directivas de componentes

Como se mencionó anteriormente, las directivas de componente son una clase. Si la combinamos con un decorador @Component es un componente o con el decorador @NgModule un módulo. A su vez Las directivas de componentes se subdividen en:

1. **Directivas de Angular** (Angular Directives): @Component, @Module, etc.

2. **Directivas customizadas** (Custom Directives): Son directivas creadas a demanda por los desarrolladores mediante la instrucción: ng g directives [nombre-de-la-directiva].

Veamos la siguiente imagen que simplifica el concepto de que una clase junto con el respectivo decorador definen la funcionalidad de la clase, si será un componente o un módulo:



Databinding en Angular

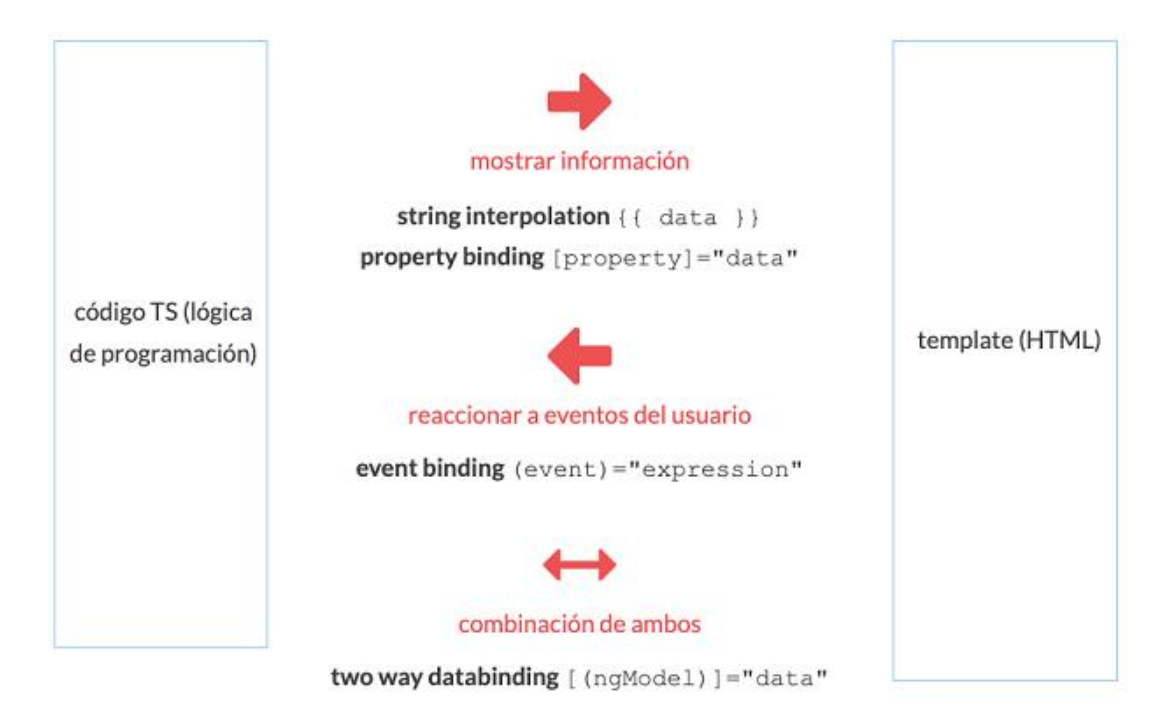
El databinding es la forma que tiene Angular para permitirnos mostrar contenido dinámico. Podríamos traducir el databinding como "comunicación" entre nuestro código HTML (template.html) y nuestra lógica de programación (componente.ts).

El databinding, nos abstrae de la lógica get/set asociada a insertar y actualizar valores en el HTML y, de convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Antes, escribir toda esa lógica era tedioso y propenso a errores dado que debíamos trabajar con javascript o alguna librería como por Jquery.

Angular nos proporciona varias maneras de comunicación entre archivos. Estas formas nos permiten:

1. **Mostrar información en el HTML**, también conocido como template, desde el archivo .ts.
2. **Pasar información al archivo TypeScript** dada por el usuario al hacer click en un botón, cambiar un menú, completar un input, etc. Es lo que se conoce como "reaccionar a eventos del usuario".
3. **Combinar a la misma vez ambas formas** de intercambio de información.

En la siguiente imagen podemos ver las 3 formas de binding o comunicación entre la lógica (componente) y el template (HTML):



Databinding en Angular. (n.d.).

Tipos de Binding

A continuación se enumeran los tipos de binding, su correspondiente sintaxis y categoría.

Tipo	Sintaxis	Categoría
Interpolation Property Attribute Class Style	{{expression}} [target]="expression" bind-target="expression"	One-way Desde el componente hacia el DOM
Event	(target)="statement" on-target="statement"	One-way Desde el DOM hacia el componente







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Two-way	[(target)]="expression" bindon-target="expression"	Two-way En ambos sentidos

Fuente: https://angular.io/guide/binding-syntax

Event binding

Event binding es la forma de comunicación que utilizamos cuando queremos reaccionar a algún evento provocado por el usuario. Por ejemplo, queremos que algo ocurra cuando el usuario haga clic en un botón, imagen, formulario, etc.

Hay un aspecto importante del evento binding que tenemos que mencionar, y es el uso de la palabra reservada (en inglés, keyword) "$event". Si el elemento forma parte del DOM entonces podemos obtener algunas propiedades del mismo con $event.target y asi, por ejemplo acceder al contenido HTML del mismo con $event.target.innerHTML o si el elemento es del tipo input, select, o un componente podemos obtener el valor guardado con $event.target.value sin usar ngModel.

Two way data binding

Two way data binding es una manera sencilla y corta de combinar los dos tipos de data binding que hemos visto anteriormente: string interpolation y event binding.

Target de un databinding

Observa que los binding types (a diferencia de los de interpolación) poseen un target (nombre) y están entre [], () o ambos [()].

Usa:

- () para enlazar del DOM al componente.
- [] para enlazar desde el componente al DOM.
- [()] para enlazar en ambos sentidos.

El target de un databinding puede ser una propiedad, un atributo, un evento, etc. Cada elemento público de una directiva está disponible para el binding en un template como se puede observar en el siguiente ejemplo:

Tipo	Target	Ejemplo
Property	src (element property) hero (component property) ngClass (directive property)	<img [src]="heroImageUrl"> <app-hero-detail [hero]="currentHero"></app-hero-detail> <div [ngClass]="{'special': isSpecial}"></div>

Event	click (element event)	<button (click)="onSave()">Save</button>
	deleteRequest (component event)	<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>
	myClick (directiva event)	<div (myClick)="clicked=$event" clickable>click me</div>
Two-way	Event y Property	<input [(ngModel)]="name">
Attribute	Attribute	<button [attr.aria-label]="help">help</button>
Class	Class property	<div [class.special]="isSpecial">Special</div>
Style	Style property	<button [style.color]="isSpecial ? 'red' : 'green'">

Fuente: https://angular.io/guide/binding-syntax

Servicios en Angular

Los protagonistas en las aplicaciones de Angular son los componentes, ya que las aplicaciones se desarrollan en base a un árbol de componentes. Sin embargo, a medida que los objetivos sean más y más complejos, lo normal es que el código de los componentes también vaya aumentando, implementando mucha lógica.

En principio esto no sería tan problemático, pero mantener la organización del código en piezas pequeñas de responsabilidad reducida es siempre muy positivo. Además, siempre llegará el momento en el que dos o más componentes tengan que acceder a los mismos datos y hacer operaciones similares con ellos, que podrían obligar al desarrollador a repetir código. Para solucionar estas situaciones existen los servicios.

Básicamente un servicio es un proveedor de datos, que mantiene lógica de acceso a ellos y operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes, que delegaron en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos.

Es decir que los servicios:

1. Son proveedores de datos.
2. Ayudan a mantener la lógica de acceso a los mismos.
3. Proveen la operatoria del negocio.
4. Manipulan de datos en la aplicación.
5. Invocar a un servidor HTTP para consumir una API

Para crear un servicio usamos el comando "generate service", indicando a continuación el nombre del servicio a generar. Por ejemplo:

ng generate service <nombre_servicio>

Es habitual colocar el servicio dentro de un módulo en concreto, y se puede lograr desde el CLI agregando una barra "/" y el nombre del servicio, pero no agrega el servicio al código de un módulo concreto, sino que colocará el archivo en el directorio de ese módulo. Luego hay que asignar el servicio al módulo donde se quiera usar. Por ejemplo:

ng generate service <nombre_modulo>/<nombre_servicio>

Como los servicios son algo que se suele usar desde varios componentes, muchos desarrolladores optan por crearlos dentro de un módulo compartido, que puede llamarse "common", "shared" o algo parecido.

Para poder usar el servicio recién creado es necesario agregarlo a un módulo. Inmediatamente se podrá usar en cualquiera de los componentes que pertenecen a este módulo, usando el decorador del módulo (@NgModule), en el array de "providers".

Por ejemplo un servicio recién creado se verá así:

```
import { Injectable } from '@anguar/core';

@Injectable()

export class MyService {

 constructor(){}

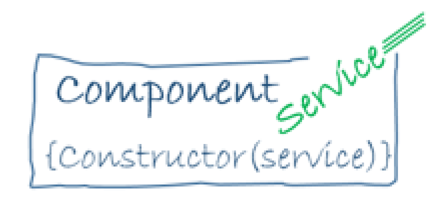
}
```

Angular CLI agrega Service al final del nombre del archivo para indicar la naturaleza de su lógica. El decorador @injectable indica a Angular que la clase que se decora, en este caso la clase MyService, puede necesitar dependencias y ser entregadas por inyección de dependencias. No todos los servicios requieren dependencias, es algo opcional de cada caso.

El import, superior, { Injectable } from '@angular/core', es importante para el funcionamiento del decorador @injectable.

¿Qué es Inyección de dependencias?

Los componentes consumen servicios; es decir, que podemos inyectar un servicio en un componente, dándole acceso al componente a ese servicio.



Inyección de dependecias. (n.d.).

De esta manera, el inyector crea dependencias y mantiene un contenedor de instancias de dependencia que utilizará si es posible. Es importante mencionar que se requiere de un proveedor, dado que éste le dice a un inyector cómo obtener o crear una dependencia.

Para cualquier dependencia que necesitemos en una aplicación, debemos registrar un proveedor con el inyector de la aplicación, con el fin de que el inyector pueda utilizar el proveedor para crear nuevas instancias. Para un servicio, el proveedor suele ser la propia clase de servicio.

La inyección de dependencias permite mantener las clases componentes ligeras y eficientes. No obtienen datos del servidor, validan la entrada del usuario o registra directamente en la consola; tales tareas son delegadas a los servicios.

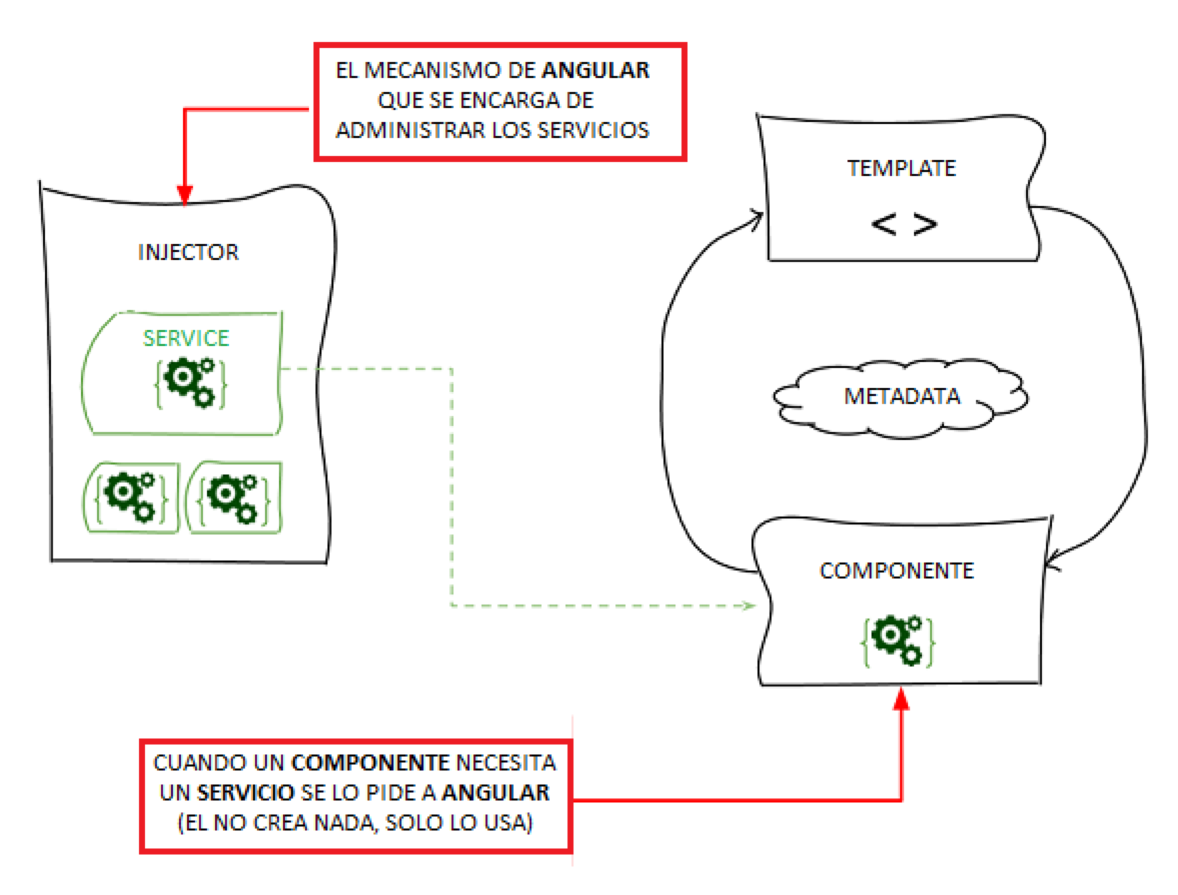
Angular, además permite la inyección de servicios de terceros, todo ello a través de la inyección de dependencias.

¿Cómo funciona la inyección de dependencias?

Cuando Angular crea una nueva instancia de un componente, determina qué servicios u otras dependencias necesita ese componente al observar los tipos de parámetros del constructor.

Si Angular descubre que un componente depende de un servicio, primero verifica si el inyector tiene instancias existentes de ese servicio. Si una instancia de servicio solicitada aún no existe, el inyector crea una utilizando el proveedor registrado y la agrega al inyector antes de devolver el servicio a Angular.

Cuando todos los servicios solicitados se han resuelto y devuelto, Angular puede llamar al constructor del componente con esos servicios como argumentos y finalmente el componente puede hacer uso del mismo.



Ciclo de inyección de dependencias. (n.d.).

Sistema de Routing en Angular

Así como en las aplicaciones web tradicionales (MPA), las aplicaciones SPA también tienen rutas, pero éstas son "virtuales" dado que el servidor no entrega una página web completa como en las MPA sino que, entrega datos, como ya expresamos anteriormente.

En Angular lo común es que el index.html en su body sólo tiene un componente raíz AppComponent y toda la acción se desarrollará en dicho componente. Todas las páginas (o vistas) se mostrarán sobre ese archivo, intercambiando el componente que se esté visualizando en cada momento. Para facilitar esto, Angular provee el sistema de routing AppRoutingModule. Éste módulo le indicará al enrutador qué vista mostrar cuando un usuario hace click en un enlace o pega una URL en la barra de direcciones del navegador.

En resumen, podemos decir que el Sistema de Routing de Angular tiene por objeto mostrar las vistas de la aplicación en función de una ruta.

Los elementos básicos que forman parte del Sistema de Rutas son:

- **El módulo del sistema de rutas**: Llamado RouterModule.
- **Rutas de la aplicación**: es un array con un listado de rutas que nuestra aplicación soportará.
- **Enlaces de navegación**: Son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.
- **Contenedor**: Donde colocar las páginas (o vistas) de cada ruta. Cada página (o vista) será representada por un componente."

Crear el módulo de rutas

Al momento de crear un proyecto de Angular, el CLI nos pregunta si queremos utilizar el módulo de rutas (Would you like to add Angular routing?), si la respuesta fue positiva los siguientes pasos serán ejecutados por el propio CLI pero si dimos una respuesta negativa tendremos que realizar de forma manual con los siguientes pasos

Para crear un módulo de rutas manualmente, ejecutar los siguientes pasos:

1. Ir a la consola o terminal de preferencia.
2. Ejecutar el comando: ng g m app-routing.module.ts
3. Editar el módulo creado anteriormente a fin de:
    1. Importar el módulo RouterModule y la clase Routes.
    2. Crear un array routes que contendrá luego las rutas virtuales.
    3. Importar y exportar el RouterModule.

A continuación te mostramos un ejemplo de cómo debería quedar finalmente el archivo app-routing.module.ts, que por el momento no tiene ninguna ruta definida:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app-routing.module.ts

En el módulo que contendrá los componentes dinámicos (app.module.ts), importar el módulo recién creado (AppRoutingModule) como te mostramos a continuación:

```
// importamos el modulo de router creado anteriormente
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { LayoutModule } from './layout/layout.module';
import { PagesModule } from './pages/pages.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    LayoutModule,
    PagesModule,
    AppRoutingModule // cargamos el modulo
      ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Módulo app.module.ts

Configurar las rutas de la aplicación

Solo nos falta definir las rutas en el array routes dónde cada ruta está configurada en base a las propiedades:

- **path**: define la ruta virtual de nuestra aplicación.
- **component**: define el componente que le dice al enrutador que componente corresponde al seleccionar dicha ruta.

En la siguiente imagen tenemos el módulo con las rutas ya configuradas:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
// Importamos nuestros componentes
import { PageOneComponent } from './components/page-one/page-one.component';
import { PageTwoComponent } from './components/page-two/page-two.component';
import { PageAboutComponent } from './components/page-about/page-
about.component';

// Definimos nuestras rutas
const routes: Routes = [
  {path: 'inicio', component:PageOneComponent},
  {path: 'otra-pagina', component: PageTwoComponent},
  {path: 'sobre-nosotros', component:PageAboutComponent},
];

@NgModule({
  // Cargamos nuestras rutas
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app-routing.module.ts

Existen propiedades complementarias que son usadas por las rutas al momento del enrutamiento como por ejemplo pathMatch para configurar que la ruta coincida parcial o total. Para consultar todas las propiedades del objeto Route se puede visitar la documentación oficial.

Hasta el momento las rutas creadas son accesibles desde el navegador, pero si quisiéramos que sean linkeables desde un template deberíamos agregar a nuestros templates el selector roterLink para la directiva RouterLink que convierte los clics del usuario en navegaciones del enrutador.

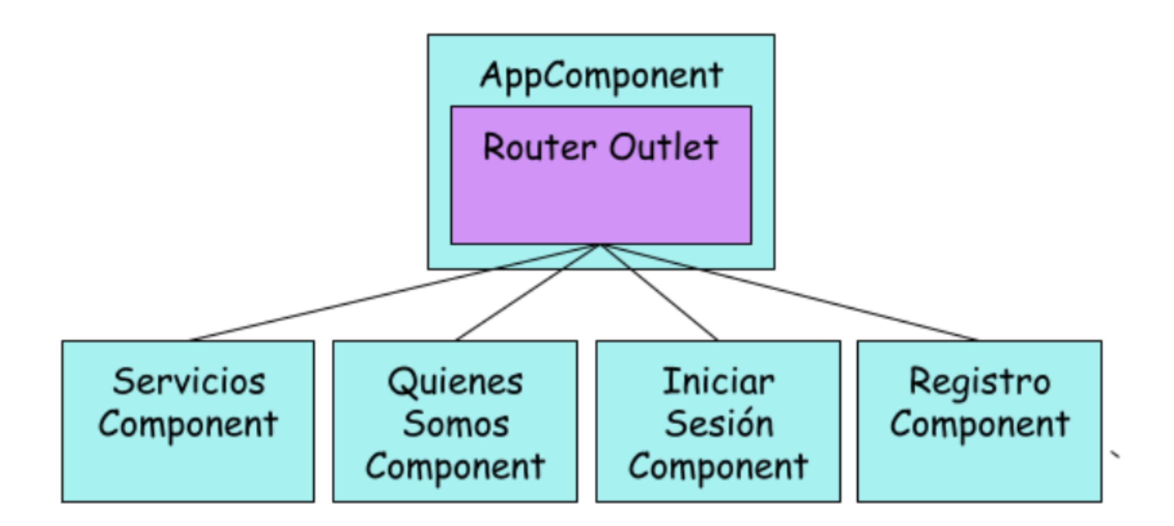
En la siguiente imagen tenemos un ejemplo de un link con el RouterLink apuntando a la URL sobre-nosotros. Al hacer click nos llevará a la ruta correspondiente, y se montará el componente asociado a la ruta:

```
<a routerLink="/sobre-nosotros" aria-current="page">About Page</a>
```

Finalmente, si queremos obtener el efecto de una SPA donde se realiza el montaje y desmontaje de componentes en base a los clicks que un usuario haga en un menú, debemos tener en cuenta la etiqueta <router-outlet></router-outlet> en el

html donde deseo que aparezca este ruteo. Esta es una etiqueta especial de Angular que sirve para mostrar los componentes hijos de un componente. Por defecto todos los componentes son hijos del componente AppComponent, por lo que, si incluimos esta etiqueta dentro de la vista de AppComponent, se renderiza cada uno de los componentes del routing dependiendo de la página en la que nos encontremos. Además, los componentes hijos pueden tener sus propios sistemas de rutas. En el caso práctico al final de esta sección hay un ejemplo de este comportamiento.

Para verlo gráficamente, si tuviéramos un componente principal,AppComponent, con su enrutador que apunta a N páginas diferentes, al cambiar de ruta, AppComponent no cambiará pero si se mostrarán/ocultarán los componentes hijos que formen parte del enrutador. En la siguiente imagen podemos ver este caso donde un AppComponente, que es un componente padre, tiene sus componentes hijos. Al incorporar el Ruter Outlet automáticamente se desprenderán todos los links a sus hijos:



Crear rutas por defecto

Siempre que tengamos una aplicación web nos interesará configurar rutas por defecto. Para configurar el redireccionamiento a un componente por defecto, como por ej. al home o inicio, hay que editar el array routes del archivo app-routing.module.ts a fin de agregar la siguiente ruta por defecto:

{path: '', redirectTo: '/inicio, pathMatch: 'full'}

Para definir una ruta por defecto, debemos utilizar 2 propiedades más de las rutas que son:

1. **redirectTo** que permite redireccionar a la ruta “/inicio” si la ruta es una cadena vacía dado

2. **patchMatch** que especifica que la coincidencia sea exacta. Es decir, que si introducimos otra cosa como ruta (que no sea la cadena vacía) no redireccionará al inicio.

Luego, si el usuario especifica la ruta vacía a nuestra web www.midominio.com, o en caso de estar en localhost http://localhost:4200/ , el sistema de routing nos llevará automáticamente a la ruta www.midominio.com/inicio o http://localhost:4200/inicio.

Crear rutas a Página 404

Sin ninguna duda, nos interesará también configurar una página 404 en el sistema de rutas para cuando el usuario intente escribir una ruta que no exista.

Previamente tendremos que tener nuestro componente de página 404, por ejemplo Pagina404 y agregar en el array del routes del app-routing.module.ts a:

{path: '**', component: Pagina404Component}

Luego, si introducimos una ruta cualquiera el sistema de rutas nos redireccionará a la página 404.

Te dejamos para probar en el caso práctico si hay alguna diferencia en agregar la ruta del 404 al inicio o final del array de routes.

Crear rutas con partes dinámicas en Angular

Las rutas pueden contener parámetros que nos interesa obtener desde los componentes. Estos parámetros son los mismos que se envían en las peticiones GET. Por ejemplo, nos gustaría que nuestra aplicación web tenga una url de tal modo que personalice la apariencia o muestre ciertos datos específicos.

Supongamos que tenemos una página de perfiles de jugadores de nuestro deporte favorito, y cada jugador tiene su ficha. La lógica nos dice que tendremos una página "/jugador" seguido de algún parámetro que puede ser un número o un string "/jugador/identificar-del-jugador" y nos quedarían algunos ejemplos así:

- /jugador/123
- /jugador/oliver-atom
- /jugador/bengie-price

Independientemente qué diseño de url hayamos elegido, la pregunta es: ¿Cómo puedo desde un componente obtener el valor que sigue luego de /jugador/? Éstas son las rutas dinámicas y para aplicarles hay que configurar el path de nuestra ruta (app-routing.module.ts) de la siguiente manera:

{path: jugador/:id, component: FichaJugadorComponent}

Donde :id es el parámetro, pero podemos poner el nombre que mejor represente el valor. Por ejemplo:

{path: jugador/:nombre-jugador, component: FichaJugadorComponent}

o

{path: jugador/:numero-jugador, component: FichaJugadorComponent}

Con esto, se crean todas las rutas posibles que empiecen con /jugador/. También es posible que quisiéramos tener rutas más específicas como por ejemplo:

{path: jugador/:numero-jugador/:posicion, component: FichaJugadorComponent}

Para recuperar u obtener el valor del o los parámetros de la URL, tendremos que editar el componente a donde apunta la ruta e importar las clases ActivatedRoute y Param. Entonces debemos importar Param de la siguiente forma:

import { ActivatedRoute, Param} from '@angular/router';

Luego inyectar en el constructor el objeto ActiveRoute y luego en el método ngOnInit se suscribe al ActivetedRoute para ejecutar el método async y finalmente obtener el o los valores recibidos en el controlador como en la siguiente imagen:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';

@Component({
  selector: 'app-ficha',
  templateUrl: './ficha.component.html',
  styleUrls: ['./ficha.component.css']
})
export class FichaJugadorComponent implements OnInit {
  constructor(private rutaActiva: ActivatedRoute) {}

  ngOnInit(): void {
    this.rutaActiva.params.subscribe((params: Params) => {
      const nombre = params.nombre.toString();

      //Implemento la logica que necesite
    });
  }
}
```

En el objeto AtivatedRoute hay una propiedad "params" que es observable y que sirve para suscribirnos a cambios en los parámetros enviados al componente. Las suscripciones a los cambios en los observadores se realizan mediante el método suscribe(). Ese método recibe varios parámetros, y el que nos interesa de momento es solo el primero, que consiste en una función callback que se ejecutará con cada cambio de aquello que se está observando.

En resumen, "this.rutaActiva.params" es el observable y "this.rutaActiva.params.subscribe()" es el método para suscribirnos a los cambios. A suscribe() le enviaremos una función callback, la cual recibirá el nuevo valor, y se ejecutará cada vez que cambie.

La función callback recibirá un objeto de clase Params. Ese objeto también deberá ser importador desde @angular/route.

Rutas hijas o anidadas

Si quisiéramos tener una ruta a una sección padre y que a su vez existen rutas a las subsecciones hijas necesitaremos hacer uso de las rutas hijas (o anidadas) y así mostrar un componente dentro de la plantilla del componente padre. De esta forma, si tuviéramos que hacer una web donde ofrecer nuestros servicios,

obtendremos rutas del estilo services/programming, services/desing, services/testing

Para definir estas rutas con una o más rutas hijas hay que agregar algo más de información en el archivo app-routing.module.ts como mostramos por ejemplo en la siguiente imagen:

```
{path:'services', component: ServiciosComponent,
  children:[
    {path:'programming', component: ProgrammingComponent},
    {path:'design', component: DesingComponent},
    {path:'testing', component: TestingComponent},
  ]}
```

Luego quedaría agregar el <router-outlet></router-outlet> en el componente padre de servicios y configurar el atributo routerLink con las rutas hijas como sigue:

- ***<a routerLink="/services">Servicios***
- ***<a routerLink="/services/programming">Programación***
- ***<a routerLink="/services/desing">Diseño***
- ***<a routerLink="/services/testing">Tesing***

Sesión de Usuario con Angular

La mayoría de las aplicaciones webs integran algún tipo de apartado privado donde solo acceden los usuarios que tengan una cuenta registrada por eso es importante que veamos cómo integrar el inicio de sesión de usuarios a nuestros proyectos de Angular.

Si bien existen varias estrategias de comunicación de sesiones entre el frontend y el backend, optamos por aplicar JWT que es hasta el momento, el método que más se está usando y que aprenderemos cómo se define. Luego veremos cómo implementar JWT en Angular, sin embargo, por ahora no contamos con las herramientas para tener un backend que nos permita efectivamente manejar una sesión así que se realizará una explicación asumiendo la existencia de ciertos comportamientos de un supuesto backend y más adelante podrás volver a repasar esta unidad.

Sesiones y JWT

La seguridad es una parte importante de cada aplicación web, y los programadores debemos asegurarnos de que nuestras aplicaciones cuentan con una

autenticación segura. El proceso de autenticación cuenta de dos partes, una que depende del lado del servidor o backend, y la otra que es el front.

Para esta sección tendremos en cuenta la implementación de una arquitectura basada en JWT.

Ejemplo de un JWT, observar que hay 3 segmentos separados por un punto:
eyJhbGciOiJIUzI1Ni
IsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw
iaWF0IjoxN
TE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

¿Qué es un JSON Web Token?

Un JSON Web Token (JWT) es un token de acceso estandarizado en el RFC 7519 que permite el intercambio seguro de datos entre dos partes. Contiene toda la información importante sobre una entidad (usuario), lo que implica que no hace falta consultar una base de datos ni que la sesión tenga que guardarse en el servidor (sesión sin estado).

Por este motivo, los JWT son especialmente populares en los procesos de autenticación. Con este estándar es posible cifrar mensajes cortos, dotarlos de información sobre el remitente y demostrar si este cuenta con los derechos de acceso requeridos. Los propios usuarios solo entran en contacto con el token de manera indirecta: por ejemplo, al introducir el nombre de usuario y la contraseña en un formulario de inicio de sesión. La comunicación como tal entre las diferentes aplicaciones se lleva a cabo en el lado del cliente y del servidor.

¿Cómo se estructura un JSON Web Token?

Un JWT firmado consta de tres partes, todas ellas codificadas en Base64 y separadas por un punto: HEADER.PAYLOAD.SIGNATURE

¿Qué significan estas tres partes?

Header

El header consta generalmente de dos valores y proporciona información importante sobre el token. Contiene el tipo de token y el algoritmo de la firma y/o cifrado utilizados. Este podría ser un ejemplo de header de un JWT: { "alg": "HS256", "typ": "JWT" }

En el ejemplo anterior, el header indica que HMAC-SHA256, abreviado como HS256, se utiliza para firmar el token. Otros métodos de cifrado típicos son RSA, con SHA-256 (RS256), y ECDSA, con SHA-256 (ES256).

Payload

El campo payload de JSON Web Token contiene la información real que se transmitirá a la aplicación. Aquí se definen algunos estándares que determinan qué datos se transmiten y cómo. La información se proporciona como pares key/value (clave-valor); las claves se denominan claims en JWT.

Hay tres tipos diferentes de claims:

1. Los **claims registrados** son los que figuran en el IANA JSON Web Token Claim Register y cuyo propósito se establece en un estándar. Algunos ejemplos son el emisor del token (iss, de issuer), el dominio de destino (aud, de audience) y el tiempo de vencimiento (exp, de expiration time). Se utilizan nombres de claim cortos para abreviar el token lo máximo posible.
2. Los **claims públicos** pueden definirse a voluntad, ya que no están sujetos a restricciones. Para que no se produzcan conflictos en la semántica de las claves, es necesario registrar los claims públicamente en el JSON Web Token Claim Register de la IANA o asignarles nombres que no puedan coincidir.
3. Los **claims privados** están destinados a los datos que intercambiamos especialmente con nuestras propias aplicaciones. Si bien los claims públicos contienen información como nombre o correo electrónico, los claims privados son más concretos. Por ejemplo, suelen incluir datos como identificación de usuario, rol, o datos privados. Al nombrarlos, es importante asegurarse de que no vayan a entrar en conflicto con ningún claim registrado o público.

Todos los claims son opcionales, por lo que no es obligatorio utilizar todos los claims registrados. En general, el payload puede contener un número ilimitado de claims, aunque es aconsejable limitar la información del JWT al mínimo. Cuanto más extenso sea el JWT, más recursos necesitará para la codificación y la descodificación.

Un payload podría estructurarse, por lo tanto, de la siguiente manera:

{ "sub": "123", "name": "Alicia", "exp": 30 }

Firma

La firma de un JSON Web Token se crea utilizando la codificación Base64 del header y del payload, así como el método de firma o cifrado especificado. La estructura viene definida por JSON Web Signature (JWS), un estándar establecido en el RFC 7515. Para que la firma sea eficaz, es necesario utilizar una clave secreta que sólo conozca la aplicación original.

Por un lado, la firma verifica que el mensaje no se ha modificado por el camino, y por otro, si el token está firmado con una clave privada, también garantiza que el remitente del JWT sea el correcto.







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Existen diferentes métodos de firma, dependiendo del nivel de confidencialidad de los datos:

1. **Sin protección**: Si los datos no requieren un nivel de protección alto, puede especificarse el valor none en el header. En este caso, no se genera ninguna firma y el JWT solo consta de header y payload. Sin esta medida de protección, el payload puede leerse como texto en claro una vez descifrado el código Base64 y no se comprueba si el mensaje procede del remitente correcto o si fue modificado al transferirse.
2. **Firma (JWS)**: por lo general, basta con comprobar si los datos provienen del remitente correcto y si han sido modificados. Para ello, se utiliza el esquema JSON Web Signature (JWS), que garantiza que el mensaje no se haya cambiado por el camino y proceda del remitente correcto. Con este procedimiento, el payload también puede leerse como texto en claro tras el descifrado de Base64.
3. **Firma (JWS) y cifrado (JWE)**: además de JWS, es posible emplear JSON Web Encryption (JWE). JWE cifra el contenido del payload, que luego se firma con JWS. Para descifrar el contenido, se indica una contraseña común o una clave privada. De este modo, el remitente se verifica, el mensaje es confidencial y se autentifica, y el payload no puede leerse como texto en claro tras el descifrado de Base64.

Tomando el ejemplo anterior, esta vez firmado:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFt
ZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.2BVS_Yu3X7FdBtUStpCXrvZ
Q3nSL2c9WudFikJnUdsU

Al decodicarlos usando la web https://jwt.io/ obtenemos por ejemplo:

```
header:{

  "alg": "HS256",

  "typ": "JWT"

}


payload:{
  "sub": "1234567890",

  "name": "John Doe",

  "iat": 1516239022

}
```

Quería comprobar la firma en la web veremos qué falla porque falta ingresar la clave de firma que estaría en el backend para autenticar el JWT: “argentinaprograma”

Resumiendo JWT

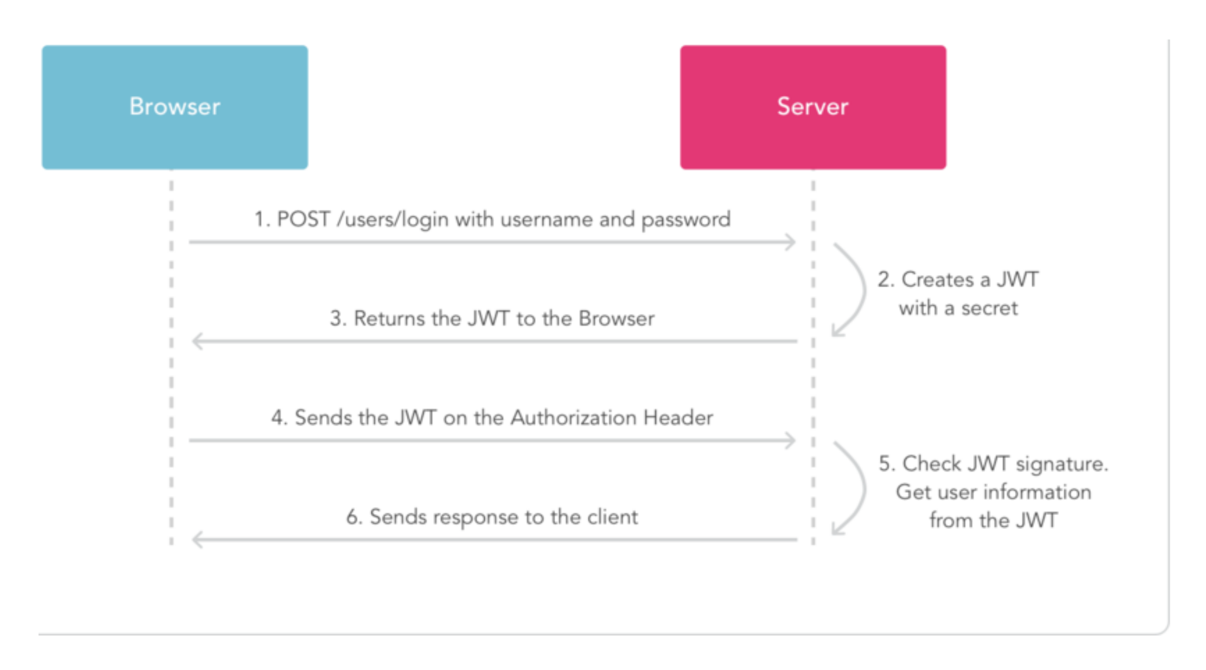
Un JSON Web Token (JWT) es básicamente un objeto representado por tres cadenas que se utilizan para transmitir información de usuario:

1. encabezamiento
2. carga útil
3. firma

Cuando un usuario inicia sesión en cualquier página web con su nombre de usuario y contraseña, el servidor de autenticación crea y envía un JWT. Este JWT se pasa junto con las subsiguientes llamadas de API al servidor. El JWT seguirá siendo válido a menos que caduque por tiempo o el usuario cierre sesión en la aplicación.

Este proceso se puede ilustrar en el diagrama a continuación donde vemos los siguientes 6 pasos:

1. Se envía una petición POST con los datos del usuario y contraseña al servidor
2. El servidor toma los datos, valida contra la base de datos y de estar todo en condiciones crea el JWT firmado con la clave secreta
3. Devuelve el JWT a la aplicación del front para ser almacenada en localStorage o la estrategia que se haya elegido
4. Las nuevas peticiones de la aplicación deberán tener configurado el Authorization Header con el token para acceder a las rutas que requieran autenticación
5. El servidor recibe una petición con el Authorization Header y verificará la firma del token. Si la firma es valida, obtendrá los datos del Token como por ejemplo el id del usuario si lo necesitara.
6. Devuelve al cliente la información solicitada



Flujo JWT. (n.d.).

Autenticación con JWT en Angular

A continuación veremos cómo implementar JWT en Angular, sin embargo será necesario contar un un servidor que nos permita completar la experiencia, por lo tanto, realizaremos una explicación asumiendo la existencia de ciertos comportamientos de un supuesto backend.

Cookies vs. Almacenamiento Local

Usaremos almacenamiento local para guardar los tokens. El almacenamiento local es un medio por el cual los datos se almacenan localmente y solo se pueden eliminar a través de JavaScript o borrando la memoria caché en el navegador. Los datos guardados en el almacenamiento local pueden persistir durante mucho tiempo. Las cookies, por otro lado, son mensajes que se envían desde el servidor al navegador y ofrecen un almacenamiento limitado.

Servicio Auth

El proceso de autenticación, a esta altura del capítulo debería haber salido intuitivamente, será manejado por un servicio. Entonces comenzamos creando un servicio Auth que facilitará la validación de la entrada y comunicación del usuario con el servidor. Para eso aplicamos el siguiente comando:

```
ng g service Auth
```

Vamos al archivo auth.service.ts donde escribiremos todo el código que interactúa con el servidor. Comenzaremos por definir en la clase del servicio la URL de la API REST y el token de firma como se muestra a continuación:

export class AuthService {

api = 'https://localhost:3000/api';

token;

}

A continuación, escribiremos el código que realiza una solicitud POST al servidor con las credenciales del usuario. Aquí, hacemos una solicitud a la API. Si tiene éxito, almacenamos el token en localStorage y redirigimos al usuario a la página de perfil.

El código quedaría de la siguiente forma:

```
import { Injectable } from '@angular/core';
import { HttpClientModule, HttpClient } from '@angular/common/http';
import { Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})

export class AuthService {

    uri = 'http://localhost:3000/api'; // La Url que corresponda en cada caso
    token;

    constructor(private http: HttpClient,private router: Router) { }

    login(email: string, password: string) {
        this.http.post(this.uri + '/authenticate', {email: email,password: password})
            .subscribe((resp: any) => {
            //Redireccionamos al usuario a su perfil
                this.router.navigate(['profile']);
              // Guardamos el token en localStorage
                localStorage.setItem('auth_token', resp.token);

            })
        );

    }

  // Para cerrar sesión elinimamos el token del localStorate
  logout() {
    localStorage.removeItem('token');
  }

  // Un servicio para verificar si existe la sesión
  public get logIn(): boolean {
    return (localStorage.getItem('token') !== null);
  }

}
```

En la imagen anterior también ya dejamos definidas las funciones de inicio y cierre de sesión:

1. **cerrar sesión**: borra el token del almacenamiento local
2. **logIn**: devuelve una propiedad booleana que determina si un usuario está autenticado

Ahora bien, tenemos una sesión iniciada, podríamos querer mostrar u ocultar algunos botones según la condición del usuario. En este caso usaremos el servicio creado logIn() de la siguiente forma:

```
<a  routerLink="/" routerLinkActive="active">Inicio</a>
<a  routerLink="profile" routerLinkActive="active" *ngIf="authService.logIn">Perfil</a>
<a  routerLink="login" routerLinkActive="active" *ngIf="!authService.logIn">Login</a>
<a class ="nav-link" (click)="logout()" href="#" *ngIf="authService.logIn">Logout</a>
```

El link "perfil" y "logout" solamente son visibles si el usuario se encuentra logueado, mientras que logout solamente es para el caso de un usuario que no haya iniciado sesión.

Para finalizar este simple ejemplo nos podríamos preguntar ¿Cómo se ejecuta el inicio de sesión? Para responder esto, debemos tener un componente, con su respectivo template y formulario que pida usuario y contraseña, y un botón que al oprimir llame al servicio creado anteriormente. Así quedaría el componente:

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';

import { AuthService } from '../auth.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  email = '';
  password = '';

  constructor(private authService: AuthService) {}

  Login(){
    // El servicio authService.login ya redirecciona
    // en caso de inicio de sesión positivo
    this.authService.login(this.email, this.password)
  }

  ngOnInit() { }
}
```

Conclusión

Esta fue una sencilla introducción al manejo de inicio de sesión. Sin embargo, el proceso puede y será mucho más complejo en la medida que los proyectos así lo requieran, como el acceso por roles o inicio de sesiones a través de redes sociales. Pero si aun sin estas complicaciones quisiéramos tener un método robusto y escalable de manejo de sesiones de usuario, el ejemplo anterior nos quedará demasiado simple. ¿Entonces cómo seguimos? Será necesario explorar otros conceptos de Angular como los Guard y las Interfaces pero aun así acá te dejamos un tutorial de 3 pasos de como aplicar Guard con los Route y CanActivave (https://angular.io/api/router/CanActivate) y obtener un Route que ya incluya la lógica de inicio de sesión para quedarnos como la siguiente imagen:

```
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuard], // Solo accesible si el usuario esta logeado
  },
];
```

Formularios en Angular

Al ser Angular un framework, y a diferencias de las librerías, ya vienen ciertos módulos preparados para hacernos las tareas más simples. En la actualidad casi no hay aplicaciones webs que no cuenten con al menos un formulario de contacto así que es importante que veamos cómo podemos construir nuestros propios formularios, con sus validaciones al momento de interactuar con el usuario y al ser enviados al backend.

Los formularios son una parte fundamental de las aplicaciones, permiten al usuario ingresar datos a la base de datos, realizar el login, registrarse, entre otras acciones muy necesarias.

Angular nos provee dos tipos de formularios:

1. **Basados en plantilla (Template driven):** proporcionan un enfoque basado en directivas. Los mismos, hacen foco en escenarios simples y no son tan reusables.
2. **Reactivos (Model Driven):** proporcionan un enfoque basado en modelos por lo que son más robustos, escalables, reusables y testeables.

En la siguiente tabla podemos ver una comparación entre ambos tipos de formularios:

	Reactivos	**Basados en plantillas**
Configuración	Explícita. Se crea en la clase del componente.	Implícita. Se crea a través de directivas.
Modelo de Datos	Estructurado e inmutable	No estructurado y mutable
Flujo de datos (entre vista y el modelo)	Síncrono	Asíncrono
Validación de Formularios	Por medio de funciones	Por medio de directivas

Tanto los formularios reactivos como los basados en plantillas realizan un seguimiento de los cambios de valor entre los elementos de entrada del formulario con los que interactúan los usuarios y los datos del formulario en su modelo de componente. Los dos enfoques comparten bloques de construcción subyacentes, pero difieren en cómo se crean y administran las instancias comunes de control de formularios.

Diferencias entre formularios basados en plantillas y reactivos

- Los formularios basados en plantillas usan FormsModule, mientras que los formularios reactivos usan ReactiveFormsModule.
- Los formularios basados en plantillas son asíncronos, mientras que los formularios reactivos son sincrónicos.
- En los formularios basados en plantillas, la mayor parte de la interacción se produce en la plantilla, mientras que, en los formularios controlados por reactivos, la mayor parte de la interacción se produce en el componente.

Ventajas y desventajas de los formularios basados en plantillas

Aunque los formularios basados en plantillas son más fáciles de crear, se convierten en un desafío cuando se quieren hacer pruebas unitarias, porque las pruebas requieren la presencia de un DOM.

Ventajas y desventajas de los formularios reactivos

Es más fácil escribir pruebas unitarias en formularios reactivos, ya que todo el código de formulario y la funcionalidad están contenidos en el componente. Sin embargo, los formularios reactivos requieren más implementación de código en el componente.

Es importante agregar, que en esta sección veremos los formularios reactivos dado que son mucho más robustos, escalables, mantenibles y testeables.

Formularios reactivos

Los formularios reactivos, también se conocen como formularios dirigidos por modelos. Emplean una técnica en la que los formularios se diseñan en el componente y luego se realizan los enlaces para el HTML. Proveen métodos explícitos e inmutables para administrar el estado del formulario cuyos valores cambian con el tiempo. Cada cambio de estado en el formulario retorna un nuevo valor permitiendo mantener la integridad con el modelo.

Además, cada elemento de la vista está directamente enlazado al modelo mediante una instancia de FormControl. Las actualizaciones de la vista al modelo y del modelo a la vista son síncronas y no dependen de la representación en la Interfaz de Usuario.

Dichos formularios están basados en flujos de datos del tipo Observable, donde cada entrada/valor puede ser accedido de manera asíncrona.

Para que Angular, intérprete nuestros formularios como reactivos, debemos importar los módulos de Formularios y Formularios Reactivos en el archivo app.module.ts o en los módulos dónde necesites trabajar dichos formularios.

En el siguiente ejemplo vemos como quedaría el módulo principal con los módulos de formularios incorporados:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
// Importamos los modulos
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

// En este componente construiremos un formulario de login
import { LoginComponent } from './login/login.component';


@NgModule({
  // Debemos cargar los modulos FormsModule y ReactiveFormsModule a nuestro proyecto
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule ],
  declarations: [ AppComponent, LoginComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

De esta forma podemos podemos pasar al siguiente paso que es meter mano en el controlador

FormControl

Es la unidad más pequeña de un formulario reactivo como por ejemplo: un cuadro de texto, un calendario, una lista desplegable, etc. Para configurar un FormControl reactivo debemos:

1. Importar la librería en el controlador de nuestro componente:

```
import {FormControl} from '@angular/forms';
```

2. Inyectar en el constructor el formBuilder
3. En el constructor de nuestro controlador crear el grupo de controles para el formulario
4. En el template del controlador enlazar el form builder utilizando el property binding [formGroup] y los atributos formControlName con sus respectivos form control

Todavía no será funcional ya que nos falta incluir las validaciones y lógica del formulario, pero así nos quedará el controlador y el template hasta este momento:

```
import { Component, OnInit } from '@angular/core';
// importamos las librerias de formulario que vamos a necesitar
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})

export class LoginComponent implements OnInit {
  form: FormGroup;

  // Inyectar en el constructor el formBuilder
  constructor(private formBuilder: FormBuilder){
    ///Creamos el grupo de controles para el formulario de login
    this.form= this.formBuilder.group({
      password:['',[]],
      mail:['', []]
   })
  }

  ngOnInit() {}

}
```

Controlador para un formulario de inicio de sesión

```
<form [formGroup]="form" >
    <div>
        <label for="email">Email: </label>
        <input type="email" formControlName="mail">
    </div>
    <br/>
    <div>
        <label for="exampleInputPassword1" class="form-label">Password: </label>
        <input type="password" formControlName="password">
    </div>
    <br/>
    <div>
        <button   type="submit">Iniciar Sesión</button>
    </div>
</form>
```

template para un formulario de inicio de sesión asociado al controlador anterior

Validaciones de Formularios

Angular nos provee la clase Validators, la cual contiene una serie de métodos estáticos que nos permiten validar las entradas de datos comunes tales como el formato del email, valores numéricos, máximos y mínimos, cantidad mínima y máxima de caracteres, entre otros.

Métodos estáticos que provee angular:

- **min()**: realiza el control de que un número sea mayor o igual al número ingresado.
- **max()**: realiza el control de que un número sea menor o igual al número ingresado.
- **required()**: valor requerido.
- **required True()**: valor requerido como verdadero. Utilizado para la casilla de verificación.
- **email()**: valor requerido pase la verificación de valor de mail.
- **minlength()**: requiere que el valor ingresado sea de una longitud mínima según la cantidad de caracteres requeridos.
- **maxlength()**: requiere que el valor ingresado sea de una longitud máxima según la cantidad de caracteres requeridos.
- **pattern()**: requiere que el valor ingresado coincida con un patrón de expresiones regulares.
- **nullValidator()**: no realizará ninguna validación.
- **compose()**: composición de varias validaciones en una sola función.
- **composeAsync()**: composición de varias validaciones asincrónicas en una sola función.

Tipo de validaciones

Angular nos provee dos tipos de validaciones:

1. **Validadores sincrónicas**: consisten en funciones síncronas que toman una instancia de control y devuelven inmediatamente un conjunto de errores de validación o un valor nulo. Se ejecutan cuando el usuario carga una letra o número, aprieta un botón, cambia una opción, etc.
2. **Validadores asíncronas**: consisten en funciones asíncronas que toman una instancia de control y devuelven una Promesa u Observable que luego emite un conjunto de errores de validación o nulos. Esto quiere decir que la validación se hace a destiempo de las acciones del usuario

Para aplicar las validaciones se deben cargar en el constructor, dentro del array de cada campo del formulario. Al solo ejemplo de entender dónde va cada tipo de validación se deja la siguiente imagen y luego aplicaremos un caso de uso:

```
this.form= this.formBuilder.group({
    password:['',[ValidacionesSincronicas],[ValidacionesAsincronicas]],
})
```

Validaciones con Form Builder

Siguiendo nuestro ejemplo de inicio de sesión, veamos los pasos para configurar las validaciones de nuestro formulario. En este caso, vamos a trabajar con validaciones síncronas sabiendo que:

1. username, debe ser requerido para el inicio de sesión y tener entre 5 y 12 carácteres
2. mail, debe ser requerido para el inicio de sesión, así como también debe respetar el formato propio del mail.
3. password, debe ser también requerido para el inicio de sesión y debe tener al menos 8 dígitos.

Para comenzar debemos Importar la clase Validators en el controlador de nuestro formulario:

import { Validators} from '@angular/forms';

y luego armar los validadores para cada caso según las reglas que mencionamos antes para username nos quedaría:

username:['',[Validators.required, Validators.minLength(5),Validators.maxLength(12)]]

- para password nos quedaría: password:['',[Validators.required, Validators.minLength(8)]],
- para email nos quedaría: mail:['', [Validators.required, Validators.email]]

Luego, para mayor comodidad y claridad del código, crear los getter de UserName, Password y Mail para acceder a tanto los valores como sus condiciones de validación desde el template y ya tendremos la validación funcionando. Sin embargo, nos faltará informarle al usuario los errores que haya en el formulario. A fin de lograr una mejor experiencia de usuario, en template podremos crear los mensajes para el usuario y los mismos, se mostrarán dependiendo de la validación. Para ello, utilizaremos las directivas *ngIf.

Dentro de las validaciones que se hacen en el template para mostrar los mensajes de error observa las propiedades:

- **.touched:** Propiedad booleana que especifica si el form control fue tocado por el usuario.
- **.errors:** Propiedad booleana que especifica si el formulario tiene errores (falla una o más validaciones).

Estados de validación

Al establecer una o más reglas para uno o más validadores, activamos el sistema de chequeo y control del estado de cada control y del formulario en su conjunto.

Cuando un control incumple con alguna regla de validación, estas se reflejan en su propiedad errors que será un objeto con una propiedad por cada regla insatisfecha y un valor o mensaje de ayuda guardado en dicha propiedad.

Los posibles estados de una validación son:

- **valid**: el control ha pasado todos los chequeos
- **invalid**: el control ha fallado al menos en una regla.
- **pending**: el control está en medio de un proceso de validación
- **disabled**: el control está desactivado y exento de validación

Estados de modificación

Los controles, y el formulario, se someten a otra máquina de estados que monitoriza el valor del control y sus cambios.

Como en el caso de los estados de validación, el formulario también se somete a estos estados en función de cómo estén sus controles.

- **prinstine**: el valor del control no ha sido cambiado por el usuario
- **dirty**: el usuario ha modificado el valor del control.
- **tocuhed**: el usuario ha tocado el control lanzando un evento blur al salir.
- **untouched**: el usuario no ha tocado y salido del control lanzando ningún evento blur.

Uno de los casos más usados es deshabilitar el botón de envío del formulario cuando la validación de algún control falla.

Validar previo enviar el formulario

Hasta el momento hicimos las validaciones mientras el usuario interactua con los campos. El siguiente paso es hacer validaciones previo a enviar los datos al servidor.

Primero, es buena práctica configurar el evento onSubmit como en la etiqueta <form> agregar (ngSumit):

<form [formGroup]="form" (ngSubmit)="onEnviar($event)">

Luego debemos agregar la función onEnviar a nuestro controlador y quedaría de la siguiente forma:

```
onEnviar(event: Event){
    // Detenemos la propagación o ejecución del compotamiento submit de un form
    event.preventDefault;

    if (this.form.valid){
      // Llamamos a nuestro servicio para enviar los datos al servidor
      // También podríamos ejecutar alguna lógica extra
    }else{
      // Corremos todas las validaciones para que se ejecuten los mensajes de error en el template
      this.form.markAllAsTouched();
    }

  }
```

Testing en Angular

Realizar tests, como pasa en todos los lenguajes, es un mundo completamente aparte, hay muchos conceptos que aprender, maneras de realizar tests, etc. Este capítulo es una pequeña introducción al mundo del testing en Angular. Hay muchas más técnicas por ver de las que vamos a repasar a continuación, te invitamos a que las investigues y reconozcas.

Introducción

Los tests son una pieza fundamental en los proyectos de hoy en día. Si tenemos un proyecto grande será esencial tener una buena forma de poder probar la aplicación sin tener que hacerlo manualmente.

Antes de meternos de lleno en testear aplicaciones Angular, es importante saber qué tipos de tests existen:

1. **Test Unitario**: Consiste en probar unidades pequeñas (componentes, por ejemplo).
2. **Test End to End (E2E)**: Consiste en probar toda la aplicación simulando la acción de un usuario, es decir, por ejemplo para desarrollo web, mediante herramientas automáticas, abrimos el navegador, navegamos y usamos la página como lo haría un usuario normal.
3. **Test de Integración**: Consiste en probar el conjunto de la aplicación asegurando la correcta comunicación entre los distintos elementos de la aplicación. Por ejemplo, en Angular observamos cómo se comunican los servicios con la API y con los componentes.

Tests unitarios con Jasmine

Para hacer tests unitarios en Angular se suele usar Jasmine. Jasmine es un framework Javascript que no es exclusivo de Angular, se puede usar en cualquier aplicación web, para la definición de test usando un lenguaje natural entendible por todo tipo de personas.

Un test en Jasmine se lo visualiza de la siguiente manera:

```
describe("Nombre del test", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Veamos las partes que conforman este simple test:

- **describe**: Define una suite de tests, es decir, una colección de tests. Esta función recibe dos parámetros: un string con el nombre de la suite y una función donde definiremos los tests.
- **it**: Define un test en particular. Recibe como parámetro el nombre del test y una función a ejecutar por el test.
- **expect**: Lo que espera recibir el test. Es decir, con expect hacemos la comprobación del test. Si la comprobación no es cierta el test falla. En el ejemplo anterior comprobamos si true es true luego el test pasa.
- No podemos simplemente hacer la comprobaciones haciendo operaciones de comparación (===) , tenemos que usar las funciones que vienen con Jasmine que sirven para facilitar la tarea, algunos ejemplos son:

- expect(array).toContain(member);
- expect(fn).toThrow(string);
- expect(fn).toThrowError(string);
- ….

Jasmine también nos permite realizar funciones que se pueden ejecutar antes de realizar un test, o después:

1. beforeAll: Se ejecuta antes de pasar todos los tests de una suite.
2. afterAll: Se ejecuta después de pasar todos los tests de una suite.
3. beforeEach: Se ejecuta antes de cada test de una suite.
4. afterEach: Se ejecuta después de cada test de una suite.

Por ejemplo, en la siguiente imagen, tenemos la suite “Hello World” que antes de ejecutar el primer test definido mediante la función it, se llama a la función beforeEach la cual cambia el valor de la variable expected, haciendo que el test pase:

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello World";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
        .toEqual(expected);
  });
});
```

Tests unitarios con Angular

Cuando creamos un proyecto y sus componentes usando Angular CLI, también se crean unos archivos .spec.ts, y eso es porque Angular CLI se encarga por nosotros de generar un archivo para testear cada uno de los componentes. Además inicializa en el archivo .spec.ts el código necesario para empezar a probar y testear los componentes.

Supongamos que tenemos un componente para tomar notas definido de la siguiente forma:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { NotesComponent } from './notes.component';

describe('NotesComponent', () => {
  let component: NotesComponent;
  let fixture: ComponentFixture;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ NotesComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(NotesComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```


```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { NotesComponent } from './notes.component';

describe('NotesComponent', () => {
  let component: NotesComponent;
  let fixture: ComponentFixture;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ NotesComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(NotesComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```


Repasemos este fragmento de codigo:

1. Lo primero que hace es crear una suite de tests para el componente con el método describe.
2. Luego de crear la suite, crea dos variables que va a necesitar para testear los componentes, el propio componente, que lo mete en la variable

component y una variable fixture de tipo ComponentFixture del componente, la cual sirve para tener el componente, pero añadiendo más información para que sea más fácil de testear.

3. A continuación, llama al método beforeEach, con una función asíncrona (sirve para asegurar que se termina de ejecutar la función asíncrona antes de pasar un test) para crear todas las dependencias del componente, en este caso, el componente en sí. Si usáramos en el componente un servicio, habría que incluirlo también, creando una sección llamada providers (como en el app.module.ts).
4. Después vuelve a llamar a la función beforeEach, esta vez, sin ser asíncrona. Crea una instancia fixture del componente usando TestBed, el cual se encargará de inyectar las dependencias definidas anteriormente mediante configureTestingModule. Para sacar el componente en sí del fixture usa componentInstance.
5. Por último crea un test para comprobar que el componente se crea correctamente, para ello, llama a la función expect y espera a que se cree bien y tenga error mediante toBeTruthy().

Para correr los tests y ver los resultados con Angular CLI el comando es: ***ng test***

Testeando clases en Angular

Imaginemos que tenemos un servicio inyectado a un componente que queremos testear. Podemos usar varias técnicas para testear el servicio:

```
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Login component', () => {

  let component: LoginComponent;
  let service: AuthService;

  beforeEach(() => {
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    localStorage.removeItem('token');
    service = null;
    component = null;
  });


  it('canLogin returns true when the user is authenticated', () => {
    localStorage.setItem('token', '12345');
    expect(component.isLogged()).toBeTruthy();
  });


});
```

En este caso, a diferencia de la estructura que crea Angular CLI, no estamos usando TestBed. Simplemente se crea el componente y el servicio y se pasa el servicio como parámetro al componente para que se inyecte mediante la inyección de dependencias. Cuando se hace el test, simplemente se llama al método del componente y se ejecuta la comprobación.

Esta técnica puede venir bien para aplicaciones pequeñas, pero si el componente necesita muchas dependencias puede llegar a ser muy tedioso andar creando todos los servicios. Además, esto no favorece la encapsulación porque estamos creando servicios y no estamos aislando el testeo del componente.

Otro problema es que tenemos que meter a mano en el localStorage un valor para que el authService funcione y devuelva true.

Creando un servicio virtual (mockeando)

Esta vez, en lugar de usar el authService real, creamos nuestra propia clase MockAuthService dentro del propio test, la cual tendrá un método con el mismo nombre que el servicio real, pero en su lugar devuelve el valor directamente.

Como se hizo antes, creamos el componente y le pasamos el servicio, en este caso, el servicio virtual que hemos creado. Usando este método no tenemos que usar el localStorage, de esta forma, solo testeamos el componente en sí y no tenemos que depender de lo que haga el servicio internamente:

```
import {LoginComponent} from './login.component';

  class MockAuthService {
    authenticated = false;

    isAuthenticated() {
      return this.authenticated;
    }
  }

  describe('Login component', () => {

    let component: LoginComponent;
    let service: MockAuthService;

    beforeEach(() => {
      service = new MockAuthService();
      component = new LoginComponent(service);
    });

    afterEach(() => {
      service = null;
      component = null;
    });


    it('canLogin returns true when the user is authenticated', () => {
      service.authenticated = true;
      expect(component.isLogged()).toBeTruthy();
    });

  });
```

Si aun así crear el servicio virtual resulta costoso, siempre podemos extender del servicio real, sobre-escribiendo los métodos que nos interesen:

```
class MockAuthService extends AuthService {
    authenticated = false;

    isAuthenticated() {
      return this.authenticated;
    }
  }
```

También podemos sobre escribir la inyección de dependencias con nuevas clases, por ejemplo:

```
TestBed.overrideComponent(
    LoginComponent,
    {set: {providers: [{provide: AuthService, useClass: MockAuthService}]}}
  );
```

Mediante del uso de spy de Jasmine

Jasmine también ofrece la posibilidad de tomar una clase y devolver directamente lo que nos interese sin tener que ejecutar internamente sus métodos:

```
import {LoginComponent} from './login.component';
  import {AuthService} from "./auth.service";

  describe('Component: Login', () => {

    let component: LoginComponent;
    let service: AuthService;
    let spy: any;

    beforeEach(() => {
      service = new AuthService();
      component = new LoginComponent(service);
    });

    afterEach(() => {
      service = null;
      component = null;
    });


    it('canLogin returns true when the user is authenticated', () => {
      spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
      expect(component.isLogged()).toBeTruthy();
    });
  });
```

Como se puede ver, con la función spyOn de Jasmine podemos hacer que el servicio devuelva directamente true en la llamada a el nombre de función que le pasamos como parámetro al spy.

Testeando llamadas asíncronas

Si por ejemplo tenemos un test que necesita validar un método asíncrono del componente o del servicio (una llamada a una API por ejemplo), podemos hacer lo siguiente:

```
it('Should get the data', fakeAsync(() => {
  fixture.componentInstance.getData();
  tick();
  fixture.detectChanges();
  expect(component.data).toEqual('new data');
}));
```

Angular proporciona el método fakeAsync para realizar llamadas asíncronas, dejándonos acceso a la llamada a tick() el cual simula el paso del tiempo para esperar a que la llamada asíncrona se realice.

Accediendo a la vista

Para acceder a los elementos html de la vista de un componente, podemos usar su fixture:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture;
  let submitButton: DebugElement;


  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    });

    fixture = TestBed.createComponent(LoginComponent);

    component = fixture.componentInstance;

    submitButton = fixture.debugElement.query(By.css('button_submit'));

  });
});
```

En este caso, accedemos al botón HTML de la vista del componente de login mediante el debugElement del fixture, el cual nos da acceso a hacer querys de elementos HTML de la vista. Como en javascript podemos acceder o cambiar las propiedades de estos elementos:

submitButton.innerText = 'Testing the button';

El TestBed del componente nos proporciona una manera de provocar que la vista se actualice con la nueva información en el caso de que hayamos cambiado algo en el componente:

fixture.detectChanges()

Testing de llamadas http

Para testear las llamadas HTTP podemos hacer dos tests, uno para comprobar que la petición se ha realizado correctamente y otro test para verificar que la información que llega de la API es correcta. Para lo segundo podemos usar la clase MockBackend de Angular, que es capaz de simular un backend con información creada por nosotros, para que cuando el servicio realice la llamada







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

HTTP en realidad llame al MockBackend para que no tenga que hacer la llamada real y podamos comprobar la información que llega al servicio.

Crear Aplicación con Angular CLI

Recordemos que el CLI nos permite automatizar algunas tareas introduciendo algunos comandos. Estas tareas son:

1. Crear un proyecto
2. Crear un modulo
3. Crear un componente
4. Crear un servicio
5. Correr un test

Para mayor información leer la documentación oficial.